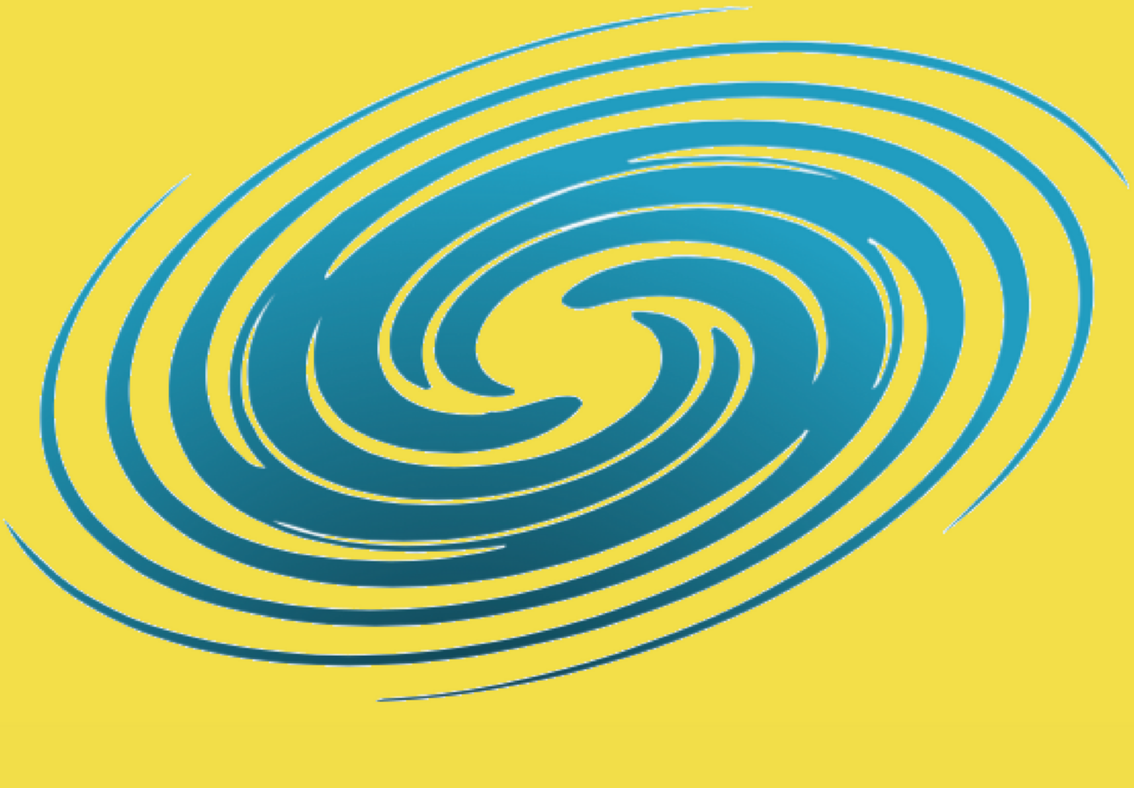
Baptiste Pesquet

THE JAVASCRIPT WAY



A modern introduction
to an essential language

The JavaScript Way

A modern introduction to an essential language

Baptiste Pesquet

This book is for sale at http://leanpub.com/thejsway

This version was published on 2017-04-09

Tweet This Book!

Please help Baptiste Pesquet by spreading the word about this book on Twitter!

The suggested hashtag for this book is #thejsway.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#thejsway

Contents

Introduction

About this book

First thing first: thanks for having chosen this book. I hope reading it will be both beneficial and pleasurable to you.

Who this book is for

This book is primarily designed for beginners. Having taught programming basics to hundreds of students, I tried to write it in the most friendly and accessible way possible. My goal was that no matter their background, everyone interested in programming should be able to follow along without too much difficulty.

However, this book can also be useful to people having some experience in software development. The JavaScript language is kind of a strange beast. It shares some similarities with other well-known programming languages such as Java or C#, starting with its syntax. On the other hand, JavaScript has *a lot* of unique characteristics that are worth learning. This book covers the majority of them. As such, it will be of interest to those wanting to get serious with JavaScript or needing their skills refreshed with the latest language evolutions.

Overview

This book is divided into three main parts. The first one teaches the basics of programming with JavaScript. The second one explains how to use JavaScript to create interactive web pages. The third one deals with more advanced concepts of the language. Each part depends on the previous ones, but there's no other prerequisite to reading.

Each chapter starts with a **TL;DR** paragraph which summarizes it, so you'll be able to skip ahead if you already know a chapter's content.

At the end of each chapter, a series of short and focused exercises will make you put your newly acquired skills into practice. Searching them seriously is *essential*: real learning comes with practicing, not just reading.

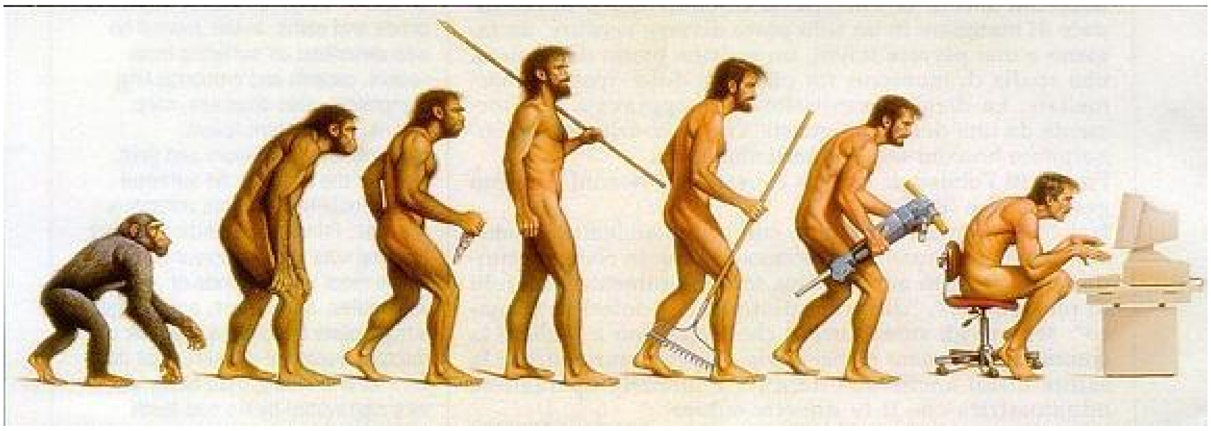
Complete beginner or already experienced in programming, I wish you a great journey in the wonderful world of JavaScript!

Welcome to programming

TL;DR

- A **computer** is a machine whose role is to execute quickly and flawlessly a series of actions given to it.
- A **program** is a list of actions given to a computer. These actions take the form of textual commands. All these commands form the program's **source code**.
- The **programmer**'s task is to create programs. To accomplish this goal, he can use different programming languages.
- Before writing code, one must think ahead and decompose the problem to be addressed in a series of elementary operations forming an **algorithm**.

What's a program?



Evolution (?)

Since their invention in the 1950s, **computers** have revolutionized our daily lives. Calculating a route from a website or a GPS, booking a train or plane ticket, or seeing and chatting with friends on the other side of the world: all these actions are possible thanks to computers.

Let's take the term "computer" in its broadest sense, meaning a machine that can perform arithmetic and logical operations. It could mean either a desktop or laptop computer (PC, Mac), a computing server, or a mobile device like a tablet or smartphone.

Nonetheless, a computer can only perform a series of simple operations when instructed to do so. They normally have no ability to learn, judge, or improvise. They simple do what they're told to do! Their value comes from how they can quickly handle and process huge amounts of information.

A computer often requires human intervention. That's where programmers and developers come in! They write programs that result in instructions to a computer.

A **computer program** (also called an application or software) is usually comprised of one or more text files containing commands in the form of code. This is why developers are also called coders.

A **programming language** is a way to give orders to a computer. It's a bit like a human language! Each programming language has vocabulary (keywords that each play a specific role) and grammar (rules defining how to write programs in that language).

How do you create programs?

Closest to the hardware: assembly language

The only programming language directly understandable by a computer is machine language, also known as **assembly language**. It is a set of very primitive operations linked to a specific family of processors (the computer's "brain") and manipulating its memory.

Here's an example of a basic program written in assembly language. It displays `"Hello"` to the user.

```
str:
    .ascii "Hello\n"
    .global _start

_start:
movl $4, %eax
movl $1, %ebx
movl $str, %ecx
movl $8, %edx
int $0x80
movl $1, %eax
movl $0, %ebx
int $0x80
```

Pretty scary, isn't it? Fortunately, other programming languages are much more simpler and convenient to use than assembly language.

Programming languages

There are a large number of programming languages, each adapted to different uses and with its own syntax. However, there are similarities between the most popular programming languages. For example, here's a simple program written in Python:

```
print("Hello")
```

You can also write the same thing in PHP:

```
<?php
    echo("Hello\n");
?>
```

Or even C#!

```
class Program {
    static void Main(string[] args) {
        Console.WriteLine("Hello");
    }
}
```

What about Java?

```
public class Program {
    public static void main(String[] args) {
        System.out.println("Hello");
    }
}
```

All these programs display `"Hello"` through a different set of instructions.

Program execution

The fact of asking a computer to process the orders contained in a program is called **execution**. Regardless of which programming language is used, a program must be translated into assembly code in order to be executed. The translation process depends on the language used.

With some languages, the translation into assembly code happens line by line in real time. In this case, the program is executed like a human reads a book, starting at the top and working down line-by-line. These languages are said to be **interpreted**. Python and PHP are examples of interpreted languages.

Another possibility is to read and check for errors the whole source code before execution. If no errors are detected, an executable targeting one specific hardware platform is generated. The intermediate step is called **compilation**, and the programming language which use it are said to be **compiled**.

Lastly, some languages are pseudo-compiled in order to be executed on different hardware platforms. This is the case for the Java language and also for those of the Microsoft .NET family (VB.NET, C#, etc).

Learn to code

Introduction to algorithms

Except in very simple cases, you don’t create programs by writing source code directly. You’ll first need to think about the instructions you’ll want to convey.

Take a concrete example from everyday life: I want to make a burrito. What are the steps that will enable me to achieve my goal?

```
Begin
    Get out the rice cooker
    Fill it with rice
    Fill it with water
    Cook the rice
    Chop the vegetables
    Stir-fry the vegetables
    Taste-test the vegetables
        If the veggies are good
            Remove them from the stove
        If the veggies aren't good
            Add more pepper and spices
        If the veggies aren't cooked enough
            Keep stir-frying the veggies
    Heat the tortilla
    Add rice to tortilla
    Add vegetables to tortilla
    Roll tortilla
End
```

Mmmmmm!

You reach your goal by combining a set of actions in a specific order. There are different types of actions:

- Simple actions (“get out the rice cooker”)
- Conditional actions (“if the veggies are good”)
- Actions that are repeated (“keep stir-frying the veggies”)

We used a simple writing style, not a specific programming language. In fact, we just wrote what is called an **algorithm**. We can define an algorithm as an ordered sequence of operations for solving a given problem. An algorithm breaks down a complex problem into a series of simple operations.

The role of the programmer

Writing programs that can reliable perform expected tasks is a programmer’s goal. A beginner can learn to quickly create simple programs. Things get more complicated when the program evolves and becomes more complex. It takes experience and a lot of practice before you feel like you’ll control this complexity! Once you have the foundation, the only limit is your imagination!

> “The computer programmer is a creator of universes for which he alone is the lawgiver. No playwright, no stage director, no emperor, however powerful, has ever exercised such absolute authority to arrange a stage or field of battle and to command such unswervingly dutiful actors or troops.” (Joseph Weizenbaum)

Introducing JavaScript

TL;DR

- Originally created to animate web pages, the JavaScript language can now bu used almost everywhere, from servers to mobile apps and connected devices.
- JavaScript is becoming essential for many software developers. It's an excellent choice as a first language for learning programming.
- It's been standardized under the name **ECMAScript** and is continuously improved ever since.
- The JavaScript version used in this book is **ES2015**, otherwise known as **ES6**. Albeit recent, it is now well supported by most environments.

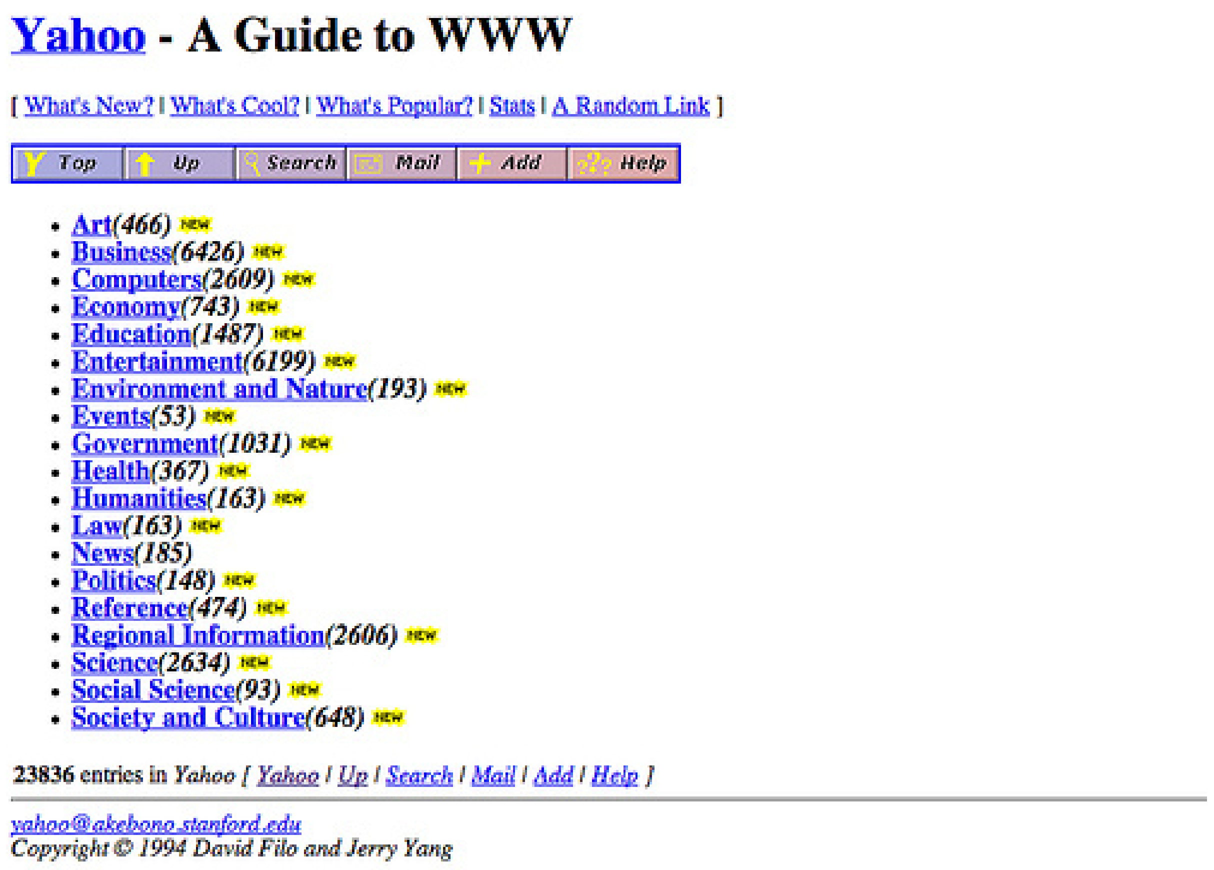
History of JavaScript

JavaScript is first and foremost the programming language of the web. It was invented in 1995 by Brendan Eich[1], who at the time worked for Netscape[2], which created the first popular web browser (Firefox's ancestor).

JavaScript should not be confused with Java, another language invented at the same time! Both share a similar syntax, but their use cases and "philosophies" are very different.

The idea behind JavaScript was to create a simple language to make web pages dynamic and interactive, since back then, pages were very simple.

[1]https://en.wikipedia.org/wiki/Brendan_Eich
[2]https://en.wikipedia.org/wiki/Netscape_Communications



Yahoo - A Guide to WWW

[What's New? | What's Cool? | What's Popular? | Stats | A Random Link]

- Art(466) NEW
- Business(6426) NEW
- Computers(2609) NEW
- Economy(743) NEW
- Education(1487) NEW
- Entertainment(6199) NEW
- Environment and Nature(193) NEW
- Events(53) NEW
- Government(1031) NEW
- Health(367) NEW
- Humanities(163) NEW
- Law(163) NEW
- News(185)
- Politics(148) NEW
- Reference(474) NEW
- Regional Information(2606) NEW
- Science(2634) NEW
- Social Science(93) NEW
- Society and Culture(648) NEW

23836 entries in Yahoo [Yahoo | Up | Search | Mail | Add | Help]

yahoo@akebono.stanford.edu
Copyright © 1994 David Filo and Jerry Yang

Yahoo's home page circa 1994

Web builders starting gradually enriching their pages by adding JavaScript code. For this code to work, the recipient web browser (the software used to surf the web) had to be able to process JavaScript. This language has been progressively integrated into browsers, and now all browsers are able to handle it!

Because of the explosion of the Web and the advent of the web 2.0 (based on rich, interactive pages), JavaScript has become increasingly popular. Web browser designers have optimized the execution speed of JavaScript, which means it's now a very fast language.

This led to the 2009 emergence of the Node.js[3] platform, which allows you to create JavaScript web applications very quickly. Thanks to a service called MongoDB[4], JavaScript has even entered the database world (software whose role is to store information).

Finally, the popularity of smartphones and tablets with different systems (iOS, Android, Windows Phone) has led to the emergence of so-called cross-platform development tools. They allow you to write a single mobile application that's compatible with these systems. These tools are almost always based on… JavaScript!

[3]https://nodejs.org
[4]https://www.mongodb.com

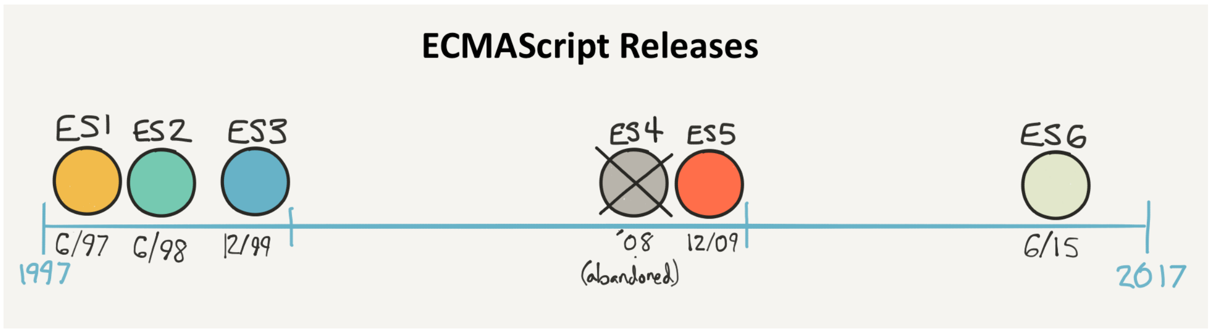
JavaScript: an essential language

In short, JavaScript is everywhere. It sits on top of a rich ecosystem of **components** (small software *bricks* that you can easily plug into your project) and a vibrant developer community. Knowing it will open the doors of the web browser-side programming (known as front-end development), server side development (backend), and mobile development. A growing number of people see JavaScript as the most important technology in software development nowadays.

Both ubiquitous and still relatively easy to learn, JavaScript is also a great choice[5] as a first language for learning programming.

Version used in this book

JavaScript was standardized in 1997 under the name ECMAScript[6]. Since then, the language has undergone several rounds of improvements to fix some awkwardness and support new features.



ECMAScript/JavaScript versions timeline

This book uses the most recently standardized version of JavaScript, called **ES2015** or sometimes **ES6**. This version brings a lot of interesting novelties to the table. It is now well supported by most environments and platforms, starting with web browsers (more details in this compatibility table[7]).

[5]https://medium.freecodecamp.com/what-programming-language-should-i-learn-first-%CA%87d%C4%B1%C9%B9%C9%94s%C9%90%CA%8C%C9%90%C9%BE-%C9%B9%C7%9D%CA%8Dsu%C9%90-19a33b0a467d#.3yu73z1px

[6]https://en.wikipedia.org/wiki/ECMAScript

[7]http://kangax.github.io/compat-table/es6/

What you'll need

TL;DR

- The easiest way to get your feet wet and code along is to use a JavaScript playground like CodePen[8].
- If you don't have access to a reliable Internet connection, an alternative is to set up a development machine with a code editor and a appropriate folder structure.
- In all cases, you'll need the most recent version of a modern web browser.

Environment setup

Few things are more frustrating than starting to learn something new and then having to spend hours trying to set up a proper work environment. Fortunately, one of the beauties of JavaScript code is that it can run on almost any browser. We'll be using this to our advantage throughout this book. No complex setup requiring administrative rights on your machine: all you'll need is a modern browser and an active Internet connection (see below if you don't have access to a reliable connection or must work offline).

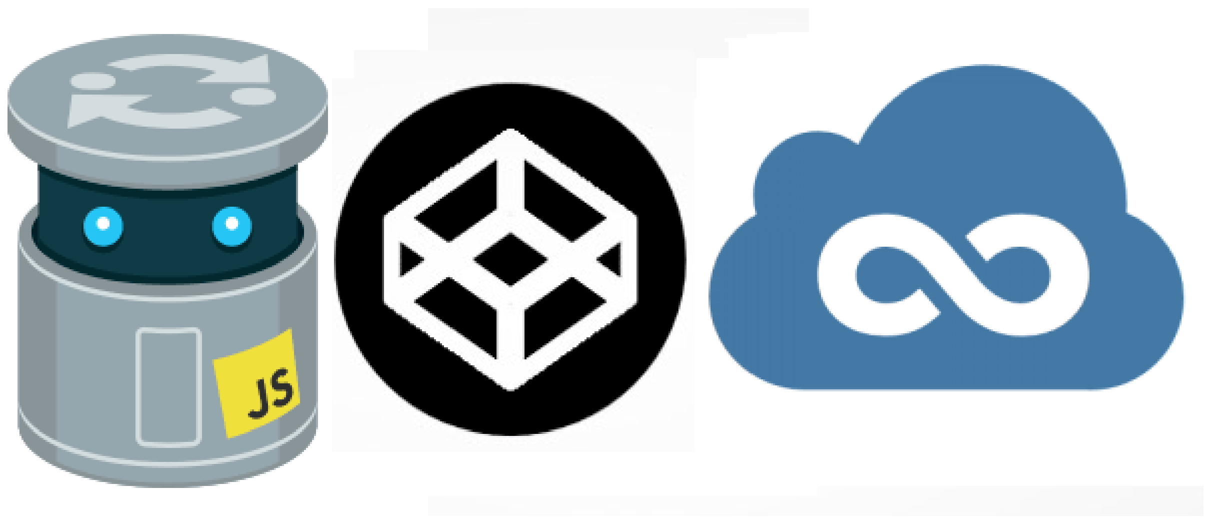
A **browser** is the software you use to visit webpages and use web applications. Check the site whatbrowser.org[9] for more info and advice about upgrading your browser.

This book targets a recent version of the JavaScript language. Whatever browser you choose to use (or have access to), it's important that you upgrade it to its most recent version.

Once your browser is updated, the quickest and easiest way to follow along is to use an online JavaScript playground. My personal favorite is CodePen[10], but there are alternatives like JSFiddle[11] and JS Bin[12].

A **JavaScript playground** is an online service where you can type some JavaScript code and immediatly visualize its result without any environment setup.

[8]http://codepen.io
[9]http://whatbrowser.org/
[10]http://codepen.io
[11]https://jsfiddle.net/
[12]http://jsbin.com/


The JSFiddle, CodePen and JS Bin logos

CodePen 101

If you choose to use CodePen along with this book, you *really* should start by visiting Welcome to CodePen[13]. It introduces the platform in a very friendly way and gives you everything you need to get started.

In addition, there are some helpful articles in the CodePen documentation about autocomplete[14], the console[15], pen autosaving[16], keybindings[17] and auto-updating[18]. Albeit not mandatory, mastering CodePen will make you more productive while studying this book.

I advise you to enable autosave and disable auto-update for all your book-related pens. Showing the CodePen console will often be needed to look at the results produced by the code.

You should use a pen (not necessarily saved) to try every code sample this book contains. You should also dedicate a specific and saved pen to each exercise you'll search.

Working offline

Using CodePen or another JavaScript playground requires a reliable Internet connection. If that's not your case (or if you prefer/have to work offline), you'll need to use a code editor on your machine. Here are some of them:

[13]https://codepen.io/hello/
[14]https://blog.codepen.io/documentation/editor/autocomplete/
[15]https://blog.codepen.io/documentation/editor/console/
[16]https://blog.codepen.io/documentation/editor/autosave/
[17]https://blog.codepen.io/documentation/editor/key-bindings/
[18]https://blog.codepen.io/documentation/editor/auto-updating-previews/

- Visual Studio Code[19] (my personal favorite).
- Brackets[20].
- Atom[21].
- Sublime Text[22].

To learn how to test JavaScript code locally and set up your machine with an appropriate folder structure, follow these instructions[23].

[19]https://code.visualstudio.com/
[20]http://brackets.io/
[21]https://atom.io/
[22]https://www.sublimetext.com/
[23]https://openclassrooms.com/courses/learn-the-basics-of-javascript/configure-your-work-environment#/id/r-3677157

I Learn to code programs

1. 3, 2, 1… Code!

Let's get started! This chapter will introduce you to the fundamentals of programming including values, types, and program structure.

TL;DR

- A **value** is a piece of information. The **type** of a value defines its role and the operations applicable to it.
- The JavaScript language uses the **number** type to represent a numerical value (with or without decimals) and the **string** type to represent text.
- A string value is surrounded by a pair of single quotes (`'...'`) or a pair of quotation marks (`"..."`).
- Arithmetic operations between numbers are provided by the `+`, `-`, `*` and `/` operators. Applied to two strings, the `+` operators joins them together. This operation is called **concatenation**.
- A computer program is made of several **lines of code** read sequentially during execution.
- **Comments** (`// ...` or `/* ... */`) are non-executed parts of code. They form a useful program documentation.
- The JavaScript command `console.log()` displays a message.

Your first program

Here's our very first JavaScript program.

```
console.log("Hello from JavaScript!");
```

This program displays the text `"Hello from JavaScript!"` in the **console**, a zone displaying textual information available in most JavaScript environnements, such as browsers.

To achieve this, its uses a JavaScript command named `console.log()`, which role is to display a piece of information. The text to be displayed is placed between parenthesis and followed by a semicolon, which mark the end of the line.

Displaying a text on the screen (the famous Hello World[1] all programmers know) is often the first thing you'll do when you learn a new programming language. It's the classic example. You've already taken that first step!

[1] https://en.wikipedia.org/wiki/Hello_world

Values and types

A **value** is a piece of information used in a computer program. Values exist in different forms called types. The **type** of a value determines its role and operations available to it.

Every computer language has its own types and values. Let's look at two of the types available in JavaScript.

Number

A **number** is a numerical value (thanks Captain Obvious). Let's go beyond that though! Like mathematics, you can use integer values (or whole numbers) such as 0, 1, 2, 3, etc, or real numbers with decimals for greater accuracy.

Numbers are mainly used for counting. The main operations you'll see are summarized in the following table. All of them produce a number result.

Operator	Role
`+`	Addition
`-`	Substraction
`*`	Multiplication
`/`	Division

String

A **string** in JavaScript is text surrounded by quotation marks, such as `"This is a string"`.

You can also define strings with a pair of single quotes: `'This is another string'`. The best practice for single or double quotes is a whole political thing. Use whichever you like, but don't mix the two in the same program!

Always remember to close a string with the same type of quotation marks you started it with.

To include special characters in a string, use the `\` character (*backslash*) before the character. For example, type `\n` to add a new line within a string: `"This is\na multiline string"`.

You can not add or remove string values like you'd do with numbers. However, the `+` operator has a special meaning when applied to two string values. It will join the two chains together, and this operation is called a **concatenation**. For example, `"Hel" + "lo"` produces the result `"Hello"`.

Program structure

We already defined a computer program as a list of commands telling a computer what to do. These orders are written as text files and make up what's called the "source code" of the program. The lines of text in a source code file are called **lines of code**.

The source code may include empty lines: these will be ignored when the program executes.

Statements

Each instruction inside a program is called a **statement**. A statement in JavaScript usually ends with a **semicolon** (albeit it's not strictly mandatory). Your program will be made up of a series of these statements.

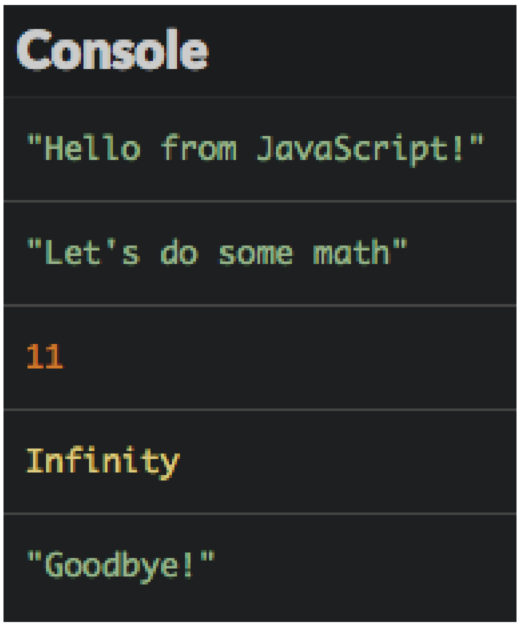
You usually write only one statement per line.

Execution flow

When a program is executed, the statements in it are "read" one after another. It's the combination of these individual results that produces the final result of the program.

Here's an example of a JavaScript program including several statements.

```
console.log("Hello from JavaScript!");
console.log("Let's do some math");
console.log(4 + 7);
console.log(12 / 0);
console.log("Goodbye!");
```



```
Console
"Hello from JavaScript!"
"Let's do some math"
11
Infinity
"Goodbye!"
```

Execution result

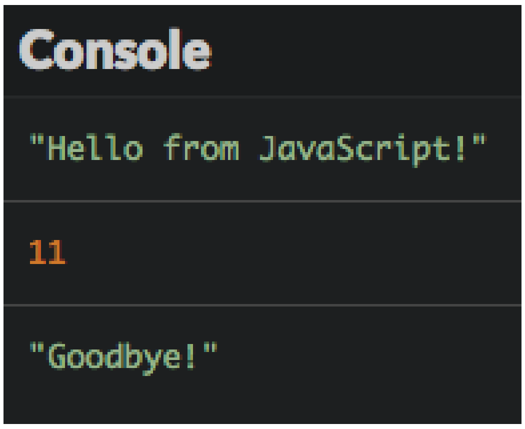
As expected, a division by zero (`12/0`) results in an `Infinity` value.

Comments

By default, each line of text in the source files of a program is considered a statement that should be executed. You can prevent certain lines from executing by putting a double slash before them: `//`. This turns the code into a **comment**.

```
console.log("Hello from JavaScript!");
// console.log("Let's do some math");
console.log(4 + 7);
// console.log(12 / 0);
console.log("Goodbye!");
```

During execution, the commented-out lines no longer produce results. As we hoped, they weren't executed.



```
Console
"Hello from JavaScript!"
11
"Goodbye!"
```

Execution result

Comments are great for developers so you can write comments to yourself, explanations about your code, and more, without the computer actually executing any of it.

You can also write comments by typing `/* */` around the code you want commented out.

```
/* A comment
written on
several lines */

// A one line comment
```

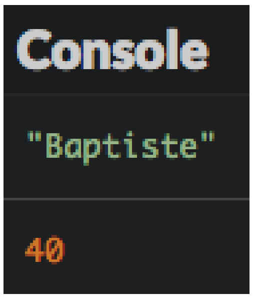
Comments are a great source of info about a program's purpose or structure. Adding comments to complicated or critical parts is a good habit you should build right now!

Coding time!

Let's put your brand new coding skills into practice.

Presentation

Write a program that displays your name and age. Here's the result for mine.



```
"Baptiste"
40
```

Minimalistic calculator

Writea a program that displays the results of adding, substracting, multiplicating and dividing 6 by 3.

Values prediction

Observe the following program and try to predict the values it displays.

```
console.log(4 + 5);
console.log("4 + 5");
console.log("4" + "5");
```

Check your prediction by executing it.

2. Play with variables

You know how to use JavaScript to display values. However, for a program to be truly useful, it must be able to store data, like information entered by a user. Let's check that out.

TL;DR

- A **variable** is an information storage area. Every variable has a **name**, a **value** and a **type**. In JavaScript, the type of a variable is deduced from the value stored in it: JavaScript is a **dynamically typed** language.
- A variable is declared using the `let` keyword followed by the variable name. To declare a **constant** (a variable whose initial value must never change), it's better to use the `const` keyword instead.
- To give a value to a variable, we use the **assignment operator** `=`. For number variables, the operators `+=` and `++` can **increment** (increase by 1) their value.
- The **scope** of a variable is the part of the program where the variable is visible. Variables declared with `let` or `const` are **block-scoped**. A **code block** is a portion of a program delimited by a pair of opening and closing curly braces `{ ... }`.
- An **expression** is a piece of code that combines variables, values and operators. Evaluating an expression produces a value, which has a type.
- Expressions may be included in strings delimited by a pair of backticks (‘). Such a string is called a **template literal**.
- **Type conversions** may happen implicitely during the evaluation of an expression, or explicitely when using the `Number()` and `String()` commands, to obtain a respectively a number or a string.
- The `prompt()` and `alert()` commands deal with information input and display under the form of dialog boxes.
- Variable naming is essential to program lisibility. Following a naming convention like camelCase[1] is good practice.

Variables

Role of a variable

A computer program stores data using variables. A **variable** is an information storage area. We can imagine it as a box in which you can put and store things!

[1]https://en.wikipedia.org/wiki/Camel_case

Variable properties

A variable has three main properties:

- Its **name**, which identifies it. A variable name may contain upper and lower case letters, numbers (not in the first position) and characters like the dollar ($) or underscore (_).
- Its **value**, which is the data stored in the variable.
- Its **type**, which determines the role and actions available to the variable.

You don't have to define a variable type explicitly in JavaScript. Its type is deduced from the value stored in the variable and may change while the program runs. That's why we say that JavaScript is a **dynamically typed** language. Other languages, like C or Java, require variable types to always be defined. This is called **static typing**.

Declaring a variable

Before you can store information in a variable, you have to create it! This is called declaring a variable. **Declaring** a variable means the computer reserves memory in which to store the variable. The program can then read or write data in this memory area by manipulating the variable.

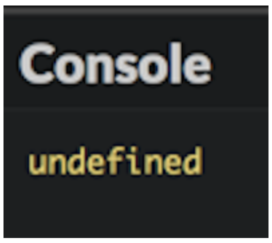
Here's a code example that declares a variable and shows its contents:

```
let a;
console.log(a);
```

In JavaScript, you declare a variable with the `let` keyword followed by the variable name. In this example, the variable created is called `a`.

In previous versions of the language, variables were declared using the `var` keyword.

Here's the execution result for this program.



```
Console
undefined
```

Execution result

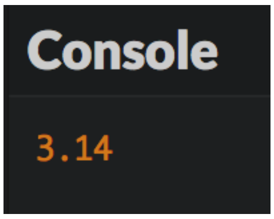
Note that the result is `undefined`. This is a special JavaScript type indicating no value. I declared the variable, calling it `a`, but didn't give it a value!

Assign values to variables

While a program is running, the value stored in a variable can change. To give a new value to a variable, use the = operator called the **assignment operator**.

Check out the example below:

```
let a;
a = 3.14;
console.log(a);
```



```
Console
3.14
```

Execution result

We modified the variable by assigning it a value. `a = 3.14` reads as "a receives the value 3.14".

Be careful not to confuse the assignment operator = with mathematical equality! You'll soon see how to express equality in JavaScript.

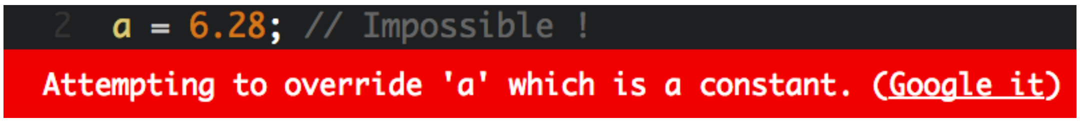
You can also combine declaring a variable and assigning it a value in one line. Just know that, within this line, you're doing two different things at once:

```
let a = 3.14;
console.log(a);
```

Declaring a constant variable

If the initial value of a variable won't ever change during the rest of program execution, this variable is called a **constant**. This constantness can be enforced by using the keyword `const` instead of `let` to declare it. Thus, the program is more expressive and further attempts to modify the variable can be detected as errors.

```
const a = 3.14; // The value of a cannot be modified
a = 6.28;       // Impossible!
```



```
a = 6.28; // Impossible !
Attempting to override 'a' which is a constant. (Google it)
```

Attempt to modify a constant variable

Increment a number variable

You can also increase or decrease a value of a number with += and ++. The latter is called an **increment operator**, as it allows incrementation (increase by 1) of a variable's value.

In the following example, lines 2 and 3 each increase the value of variable b by 1.

```
let b = 0;       // b contains 0
b += 1;          // b contains 1
b++;             // b conttains 2
console.log(b); // Shows 2
```

Variable scope

The **scope** of a variable is the part of the program where the variable is visible and usable. Variables declared with `let` or `const` are **block-scoped**: their visibility is limited to the block where they are declared (and every sub-block, if any). In JavaScript and many other programming languages, a **code block** is a portion of a program delimited by a pair of opening and closing braces. By default, a JavaScript program forms one block of code.

```
let nb1 = 0;
{
    nb1 = 1; // OK : nb1 is declared in the parent block
    const nb2 = 0;
}
console.log(nb1); // OK : nb1 is declared in the current block
console.log(nb2); // Error! nb2 is not visible here
```

Expressions

An **expression** is a piece of code that produces a value. An expression is created by combining variables, values and operators. Every expression has a value and thus a type. Calculating an expression's value is called **evaluation**. During evaluation, variables are replaced by their values.

```
// 3 is an expression whose value is 3
const c = 3;
// c is an expression whose value is the value of c (3 here)
let d = c;
// (d + 1) is an expression whose value is d's + 1 (4 here)
d = d + 1; // d now contains the value 4
console.log(d); // Show 4
```

Operator priority inside an expression is the same as in math. However, an expression can integrate **parenthesis** that modify these priorities.

```
let e = 3 + 2 * 4; // e contains 11 (3 + 8)
e = (3 + 2) * 4;   // e contains 20 (5 * 4)
```

It is possible to include expressions in a string by using **backticks** (`) to delimitate the string. Such a string is called a **template literal**. Inside a template literal, expressions are identified by the `${expression}` syntax.

This is often used to create strings containing the values of some variables.

```
const country = "France";
console.log(`I live in ${country}`); // Show "I live in France"
const x = 3;
const y = 7;
console.log(`${x} + ${y} = ${x + y}`); // Show "3 + 7 = 10"
```

Type conversions

An expression's evaluation can result in type conversions. These are called **implicit** conversions, as they happen automatically without the programmer's intervention. For example, using the + operator between a string and a number causes the concatenation of the two values into a string result.

```
const f = 100;
// Show "Variable f contains the value 100"
console.log("Variable f contains the value " + f);
```

JavaScript is extremely tolerant in terms of type conversion. However, sometimes conversion isn't possible. If a number fails to convert, you'll get the result `NaN` (*Not a Number*).

```
const g = "five" * 2;
console.log(g); // Show NaN
```

Sometimes you'll wish to convert the value of another type. This is called **explicit** conversion. JavaScript has the `Number()` and `String()` commands that convert the value between the parenthesis to a number or a string.

```
const h = "5";
console.log(h + 1); // Concatenation: show the string "51"
const i = Number("5");
console.log(i + 1); // Numerical addition: show the number 6
```

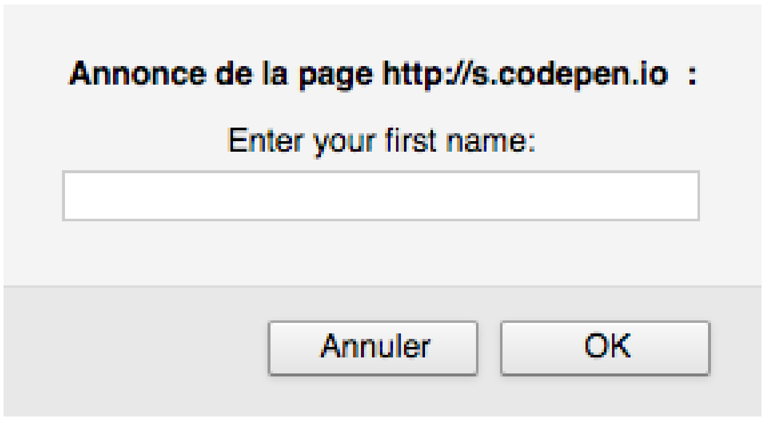
User interactions

Entering information

Once you start using variables, you can write programs that exchange information with the user.

```
const name = prompt("Enter your first name:");
alert(`Hello, ${name}`);
```

During execution, an dialog box pops up, asking for your name.

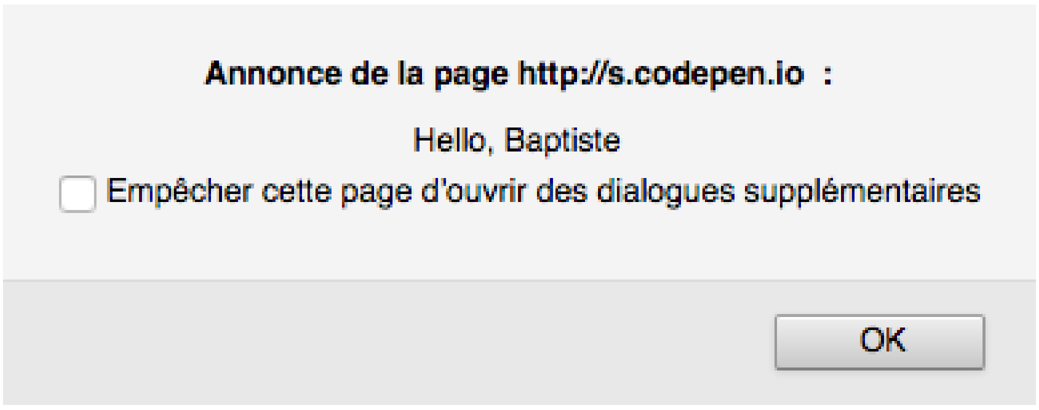

Annonce de la page http://s.codepen.io :

Enter your first name:

Execution result

This is the result of the JavaScript command `prompt("Enter your first name:")`.

Type your name and click **OK**. You'll then get a personalized greeting.


Annonce de la page http://s.codepen.io :

Hello, Baptiste

☐ Empêcher cette page d'ouvrir des dialogues supplémentaires

Execution result

The value you entered in the first dialog box has been stored as a string in the variable `name`. The JavaScript command `alert()` then triggered the display of the second box, containing the result of the concatenation of the string `"Hello, "` with the value of the `name` variable.

Displaying information

Both `console.log()` (encountered in the previous chapter) and `alert()` can be used to display information to the user. Unlike `alert()`, `console.log()` does not stop program execution and is often a better choice.

`console.log()` can also display several comma-separated values at once.

```
const temp1 = 36.9;
const temp2 = 37.6;
const temp3 = 37.1;
console.log(temp1, temp2, temp3); // Show "36.9 37.6 37.1"
```

Entering a number

Regardless of the entered data, the `prompt()` command always return a string value. If this value is to be used in numerical expressions, it *must* be converted into a number with the `Number()` command.

```
const input = prompt("Enter a number:"); // input's type is string
const nb = Number(input); // nb's type is number
```

Both operations can be combined in one line for the same result.

```
const nb = Number(prompt("Enter a number:")); // nb's type is number
```

In this example, the user input is directly converted in a number value by the `Number()` command and stored in the `nb` variable.

Variable naming

To close this chapter, let's discuss variable naming. The computer doesn't care about variable names. You could name your variables using the classic example of a single letter (`a`, `b`, `c`...) or choose absurd names like `burrito` or `puppieskittens90210`.

Nonetheless, naming variables well can make your code much easier to read. Check out these two examples:

```
const nb1 = 5.5;
const nb2 = 3.14;
const nb3 = 2 * nb2 * nb1;
console.log(nb3);
```

```
const radius = 5.5;
const pi = 3.14;
const perimeter = 2 * pi * radius;
console.log(perimeter);
```

They function in the same way, but the second version is much easier to understand.

How are some good ways to name variables?

Choose meaningful names

The most important rule is to give each variable a name that reflects its role. This is the case in the second example above, where you use the word `radius` to actually indicate…well, a radius.

Don’t use reserved words

The keywords of JavaScript are reserved names. They should not be used as variable names. Here’s the list of reserved words in JavaScript[2].

Follow a naming convention

It can take several words to describe the roles of certain variables. The most common way to account for these variables is to use the camelCase[3] naming convention, based on two main principles:

- All variable names begin with a **lowercase** letter.
- If the name of a variable consists of several words, the first letter of each word (except the first word) is **uppercase**.

⚠ Like many other languages, JavaScript is **case sensitive**. For example, `myVariable` and `myvariable` are two different variable names. Be careful!

Coding time!

Build a habit of choosing good variables name in all exercises, starting with these ones.

Improved hello

Write a program that asks the user for his first name and his last name. Th program then displays them in one sentence.

Final values

Observe the following program and try to predict the final values of its variables.

[2] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Lexical_grammar#Keywords
[3] https://en.wikipedia.org/wiki/Camel_case

```
let a = 2;
a = a - 1;
a++;
let b = 8;
b += 2;
const c = a + b * b;
const d = a * b + b;
const e = a * (b + b);
const f = a * b / a;
const g = b / a * a;
console.log(a, b, c, d, e, f, g);
```

Check your prediction by executing it.

VAT calculation

Write a program that asks the user for a raw price. After that, its calculates the corresponding final price using a VAT rate of 20.6%.

From Celsius to Fahrenheit degrees

Write a program that asks for a temperature in Celsius degrees, then displays it in Fahrenheit degrees.

> The conversion between scales is given by the formula: [°F] = [°C] x 9/5 + 32.

Variable swapping

Observe the following program.

```
let number1 = 5;
let number2 = 3;

// Type your code here (and nowhere else!)

console.log(number1); // Should show 3
console.log(number2); // Should show 5
```

Add the necessary code to swap the values of variables `number1` and `number2`.

> This exercise has several valid solutions. You may use more than two variables to solve it.

3. Add conditions

Up until now, all the code in our programs has been executed chronologically. Let's enrich our code by adding conditional execution!

TL;DR

- The `if` keyword defines a **conditional statement**, also called a **test**. The associated code block is only run if the **condition** is satisfied (its value is `true`). Thus, a condition is an expression whose evaluation always produces a boolean result (`true` or `false`).

```
if (condition) {
    // Code to run when the condition is true
}
```

- The code block associated to an `if` is delimited by a pair of opening and closing braces. To improve lisibility, its statements are generally **indented** (shifted to the right).
- The **comparison operators** `===`, `!==`, `<`, `<=`, `>` et `>=` are used to compare numbers inside a condition. All of them return a boolean result.
- An `else` statement can be associated to an `if` to express an **alternative**. Depending on the condition value, either the code block associated to the `if` or the one associated to the `else` will be run, but never both. There is no limit to the depth of condition nesting.

```
if (condition) {
    // Code to run when the condition is true
}
else {
    // Code to run when the condition is false
}
```

- Complex conditions can be created using the **logical operators** `&&` ("and"), `||` ("or") et `!` ("not").
- The `switch` statement is used to kick off the execution of one code block among many, depending on the value of an expression.

```
switch (expression) {
case value1:
    // Code to run when the expression matches value1
    break;
case value2:
    // Code to run when the expression matches value2
    break;
...
default:
    // Code to run when neither case matches
}
```

What's a condition?

Suppose we want to write a program that makes enter a number to the user, who then displays a message if the number is positive. Here the corresponding algorithm.

```
Enter a number
Si the number is positive
    Display a message
```

The message should display only if the number is positive: this means it's "subject" to a **condition**.

The `if` statement

Here's how you translate the program to JavaScript.

```
const number = Number(prompt("Enter a number:"));
if (number > 0) {
    console.log(`${number} is positive`);
}
```

The `console.log(...)` command is executed only *if* the number is positive. Test this program to see for yourself!

Conditional syntax looks like this:

```
if (condition) {
    // Code to run when the condition is true
}
```

The pair of opening and closing braces defines the block of code associated with an `if` statement. This statement represents a **test**. It results in the following: "If the condition is true, then executes the instructions contained in the code block".

The condition is always placed in parentheses after the `if`. The statements within the associated code block are shifted to the right. This practice is called **indentation** and helps make your code more readable. As your programs grow in size and complexity, it will become more and more important. The indentation value is often 2 or 4 spaces.

When the code block has only one statement, braces may be omitted. As a beginner, you should nonetheless always use braces when writing your first conditions.

Conditions

A **condition** is an expression that evaluates as a value either true or false: it's called a **boolean** value. When the value of a condition is true, we say that this condition is satisfied.

We have already studied numbers and strings, two types of data in JavaScript. Booleans are another type. This type has only two possible values: `true` and `false`.

Any expression producing a boolean value (either `true` or `false`) can be used as a condition in an `if` statement. If the value of this expression is `true`, the code block associated with it is executed.

```
if (true) {
    // The condition for this if is always true
    // This block of code will always be executed
}
if (false) {
    // The condition for this if is always false
    // This block of code will never be executed
}
```

Boolean expressions can be created using the comparison operators shown in the following table.

Operator	Meaning
`===`	Equal
`!==`	Not equal to
`<`	Less than
`<=`	Less than or equal to
`>`	Greater than
`>=`	Greater than or equal to

In some other programming languages, equality and inequality operators are `==` and `!=`. They also exist in JavaScript, but it's safer to use `===` and `!==` (more details[1]).

[1]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Equality_comparisons_and_sameness

It's easy to confuse comparison operators like === (or ==) with the assignment operator =. They're very, very different. Be warned!

Now let's modify the example code to replace > with >= and change the message, then test it with the number 0.

```
const number = Number(prompt("Enter a number:"));
if (number >= 0) {
    console.log(`${number} is positive or zero`);
}
```

If the user input is 0, the message appears in the console, which means that the condition(`number >= 0`) was satisfied.

Alternative conditions

You'll often want to have your code execute one way when something's true and another way when something's false.

The `else` statement

Let's enrich our sample with different messages depending if the number's positive or not.

```
const number = Number(prompt("Enter a number:"));
if (number > 0) {
    console.log(`${number} is positive`);
}
else {
    console.log(`${number} is negative or zero`);
}
```

Test this code with a positive number, negative number, and zero, while watching the result in the console. The code executes differently depending if the condition (`number > 0`) is true or false.

The syntax for creating an alternative is to add an `else` keyword after an initial `if`.

```
if (condition) {
    // Code to run when the condition is true
}
else {
    // Code to run when the condition is false
}
```

You can translate an `if/else` statement like this: “If the condition is true, then execute this first set of code; otherwise, execute this next set of code”. Only one of the two code blocks will be executed.

Nesting conditions

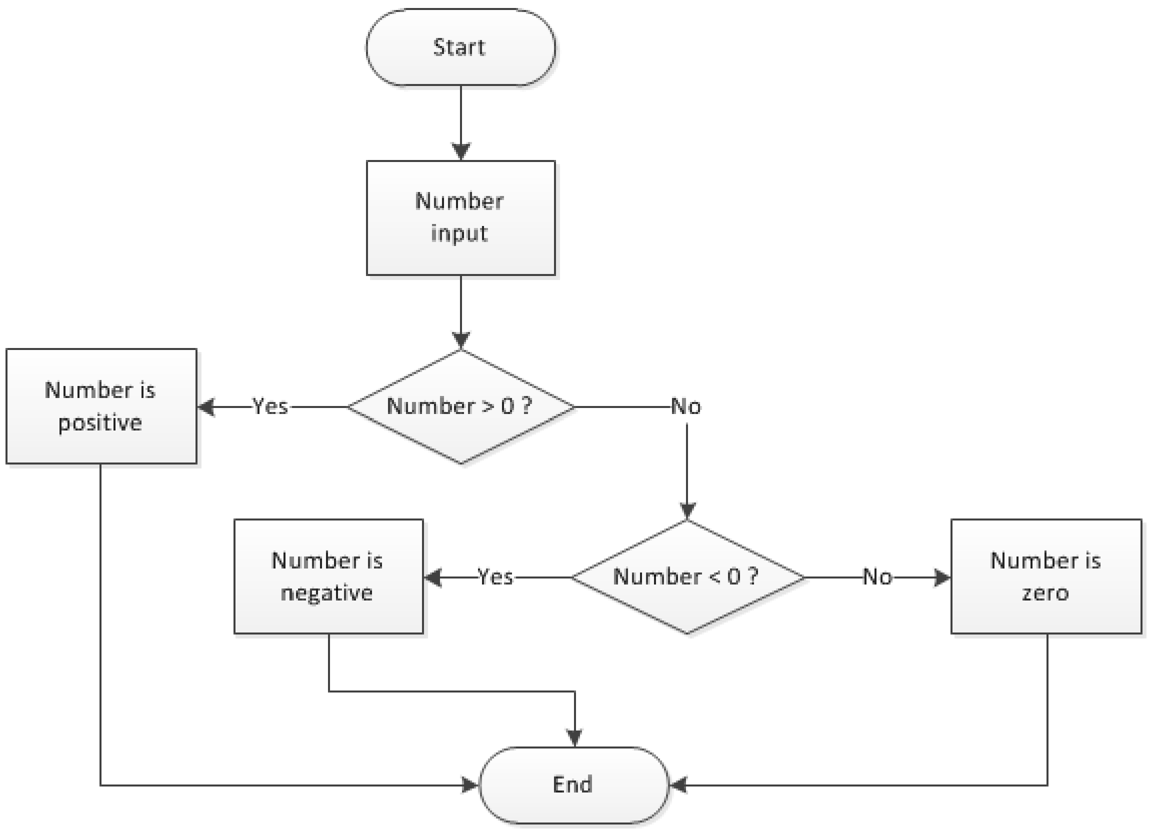
Let’s go next level and display a specific message if the entered number is zero. See this example, which has a positive test case, negative test case, and a last resort of the number being zero.

```
const number = Number(prompt("Enter a number:"));
if (number > 0) {
    console.log(`${number} is positive`);
} else { // number <= 0
    if (number < 0) {
        console.log(`${number} is nagative`);
    } else { // number === 0
        console.log(`${number} is zero`);
    }
}
```

Let’s wrap our heads around it. If the code block associated to the first `else` is run, then the number has to be either strictly negative or zero. Inside this block, a second `if` statement checks if the number is negative. If it’s not, we know for sure that it’s zero.

When learning to write nested conditions, you should add descriptive comments to each condition, just like in the previous example.

The execution flow for the previous program can be expressed graphically using a **flow diagram**.



Example flow diagram

This example shows how essential indentation is for understanding a program's flow. There is no limit to the possible depth of condition nesting, but too many will affect program lisibility.

A particular case happens when the only statement in a `else` block is an `if`. In that case, you can write this `else` on the same line as the `if` and without braces. Here's a more concise way to write our example program.

```
const number = Number(prompt("Enter a number:"));
if (number > 0) {
    console.log(`${number} is positive`);
} else if (number < 0) {
    console.log(`${number} is negative`);
} else {
    console.log(`${number} is zero`);
}
```

Add additional logic

"And" operator

Suppose you want to check if a number is between 0 and 100. You're essentially checking if it's "greater than or equal to 0" and "less than or equal to 100". Both sub-conditions must be satisfied at the same time.

The expression `0 <= nombre <= 100` is correct from a mathematical point of view but cannot be written in JavaScript (neither in most other programming languages).

Here's how you'd translate that same check into JS.

```
if ((number >= 0) && (number <= 100)) {
    console.log(`${number} is between 0 and 100, both included`);
}
```

Parentheses between sub-conditions are not mandatory but I advise you to add them anyway, to avoir nasty bugs in some special cases.

The `&&` operator ("logical and") can apply to both types of boolean values. `true` will only be the result of the statement if both conditions are true.

```
console.log(true && true);   // true
console.log(true && false);  // false
console.log(false && true);  // false
console.log(false && false); // false
```

The previous result is the **truth table** of the `&&` operator.

"Or" operator

Now imagine you want to check that a number is outside the range of 0 and 100. To meet this requirement, the number should be less than 0 or greater than 100.

Here it is, translated into JavaScript:

```
if ((number < 0) || (number > 100)) {
    console.log(`${number} is not in between 0 and 100`);
}
```

The `||` operator ("logical or") makes statements `true` if at least one of the statements is true. Here's its truth table:

```
console.log(true || true);   // true
console.log(true || false);  // true
console.log(false || true);  // true
console.log(false || false); // false
```

"Not" operator

There's another operator for when you know what you don't want: the not operator! You'll use a `!` for this.

```
if (!(number > 100)) {
    console.log(`${number} is less than or equal to 100`);
}
```

Here's the truth table of the `!` operator.

```
console.log(!true);  // false
console.log(!false); // true
```

Multiple choices

Let's write some code that helps people decide what to wear based on the weather using `if`/`else`.

```
const weather = prompt("What's the weather like?");
if (weather === "sunny") {
    console.log("T-shirt time!");
} else if (weather === "windy") {
    console.log("Windbreaker life.");
} else if (weather === "rainy") {
    console.log("Bring that umbrella!");
} else if (weather === "snowy") {
    console.log("Just stay inside!");
} else {
    console.log("Not a valid weather type");
}
```

When a program should trigger a block from several operations depending on the value of an expression, you can write it using the JavaScript statement `switch` to do the same thing.

```
const weather = prompt("What's the weather like?");
switch (weather) {
case "sunny":
    console.log("T-shirt time!");
    break;
case "windy":
    console.log("Windbreaker life.");
    break;
case "rainy":
    console.log("Bring that umbrella!");
    break;
case "snowy":
    console.log("Winter is coming! Just stay inside!");
    break;
default:
    console.log("Not a valid weather type");
}
```

If you test it out, the result will be the same as the previous version.

The `switch` statement kicks off the execution of one code block among many. Only the code block that matches the relevant situation will be executed.

```
switch (expression) {
case value1:
    // Code to run when the expression matches value1
    break;
case value2:
    // Code to run when the expression matches value2
    break;
...
default:
    // Code to run when neither case matches
}
```

You can set as many cases as you want! The word `default`, which is put at the end of `switch`, is optional. It can let you handle errors or unexpected values.

Adding a `break;` in each block is important so you get out of the switch statement!

```
const x = "abc";
switch (x) {
case "abc":
    console.log("x = abc");
    // break omitted: the next block is also run!
case "def":
    console.log("x = def");
    break;
}
```

The previous example show `"x = abc"` (the correct result) but also `"x = def"`.

Coding time!

Here are a few advice about these exercises:

- Keep on choosing your variable names wisely, and respect indentation when creating code blocks associated to `if`, `else` and `switch` statements.
- Try to find alternative solutions. For example, one using an `if` and another using a `switch`.
- Test your programs thoroughly, without fear of finding mistakes. It's a very important skill.

Following day

Write a program that accepts a day name from the user, then shows the name of the following day. Incorrect inputs must be taken into account.

Number comparison

Write a program that accepts two numbers, then compare their values and displays an appropriate message in all cases.

Final values

Take a look at the following program.

```
let nb1 = Number(prompt("Enter nb1:"));
let nb2 = Number(prompt("Enter nb2:"));
let nb3 = Number(prompt("Enter nb3:"));

if (nb1 > nb2) {
    nb1 = nb3 * 2;
} else {
    nb1++;
    if (nb2 > nb3) {
        nb1 = nb1 + nb3 * 3;
    } else {
        nb1 = 0;
        nb3 = nb3 * 2 + nb2;
    }
}
console.log(nb1, nb2, nb3);
```

Before executing it, try to guess the final values of variables `nb1`, `nb2` and `nb3` depeding on their initial values. Complete the following table.

Initial values	`nb1` final value	`nb2` final value	`nb3` final value
`nb1=nb2=nb3=4`			
`nb1=4,nb2=3,nb3=2`			
`nb1=2,nb2=4,nb3=0`			

Check your predictions by executing the program.

Number of days in a month

Write a program that accepts a month number (between 1 and 12), then shows the number of days of that month. Leap years are excluded. Incorrect inputs must be taken into account.

Following second

Write a program that asks for a time under the form of three informations (hours, minutes, seconds). The program calculates and shows the time one second after. Incorrect inputs must be taken into account.

> This is not as simple as it seems… Look at the following results to see for yourself:
>
> - 14h17m59s ⇒ 14h18m0s
> - 6h59m59s ⇒ 7h0m0s
> - 23h59m59s ⇒ 0h0m0s (midnight)

4. Repeat statements

In this chapter, we'll look at how to execute code on a repeating basis.

TL;DR

- **Loops** are used to repeat a series of statements. Each repetition is called an **iteration**. The code block associated to a loop is called its **body**.
- The `while` loop repeats statements *while* a certain condition is true. The `for` loop gives the ability to manage what happens just before the loop starts and after each loop iteration has run.

```
// While loop
while (condition) {
    // Code to run while the condition is true
}

// For loop
for (initialization; condition; final expression) {
    // code to run while the condition is true
}
```

- The variable associated to the loop condition is called the loop **counter** and often named `i`.
- Beware! The condition of a `while` loop must eventually become false, to avoid the risk of an **infinite loop**. Also, updating the counter of a `for` loop inside its body is a bad idea.
- All loops can be written with `while`, but if you know in advance how many times you want the loop to run, `for` is the best choice.

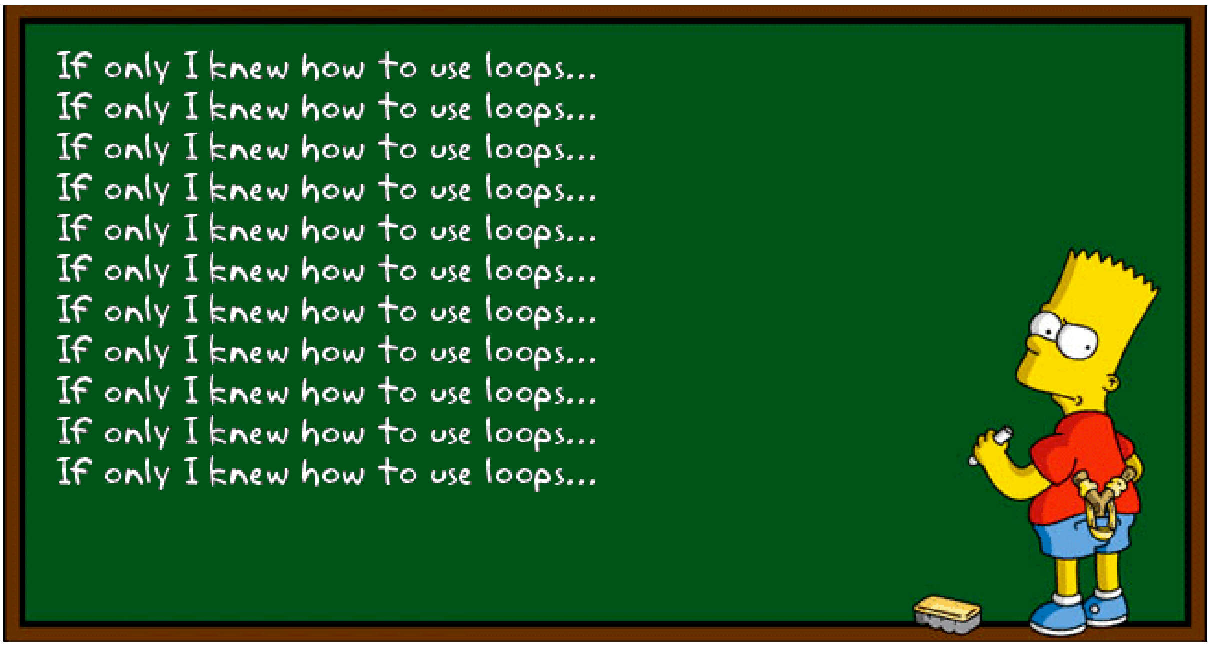
Introduction

If you wanted to write code that displayed numbers between 1 and 5, you could do it with what you've already learned:

```
console.log(1);
console.log(2);
console.log(3);
console.log(4);
console.log(5);
```

This is pretty tiresome though and would be much more complex for lists of numbers between 1 and 1000, for example. How can you accomplish the same thing more simply?

JavaScript lets you write code inside a **loop** that executes repeatedly until it's told to stop. Each time the code runs, it's called an **iteration**.


If only I knew how to use loops...
If only I knew how to use loops...
If only I knew how to use loops...
If only I knew how to use loops...
If only I knew how to use loops...
If only I knew how to use loops...
If only I knew how to use loops...
If only I knew how to use loops...
If only I knew how to use loops...
If only I knew how to use loops...
If only I knew how to use loops...

The `while` loop

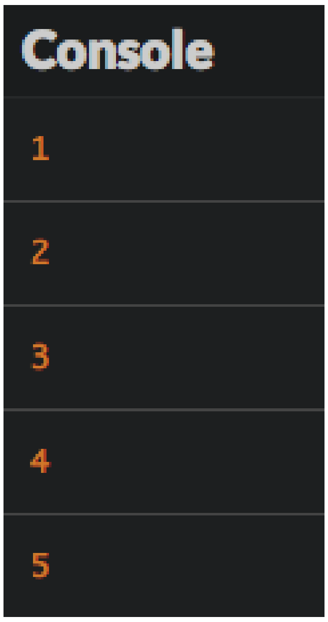
A `while` look lets you repeat code while a certain condition is true.

Example

Here's a sample program written with a `while` loop.

```
let number = 1;
while (number <= 5) {
    console.log(number);
    number++;
}
```

Just like the previous one, it shows all integer numbers between 1 and 5.



```
Console
1
2
3
4
5
```

Execution result

How it works

You’ll use the following syntax to write a `while` loop.

```
while (condition) {
    // Code to run while the condition is true
}
```

Before each loop iteration, the condition in parentheses is evaluated to determine whether it’s true or not. The code associated with a loop is called its **body**.

- If the condition’s value is `true`, the code in the `while` loop’s body runs. Afterwards, the condition is re-evaluated to see if it’s still true or not. The cycle continues!
- If the condition’s value is `false`, the code in the loop stops running or doesn’t run.

The loop body must be placed within curly braces, except if it’s only one statement. For now, always use curly braces for your loops.

The `for` loop

You’ll often need to write loops with conditions that are based on the value of a variable updated in the loop body, like in our example. JavaScript offers another loop type to account for this: the `for` loop.

Example

Here’s the same program as above written instead with a `for` loop.

```
let number;
for (number = 1; number <= 5; number++) {
    console.log(number);
}
```

It gives exactly the same result.

How it works

Here's the `for` loop syntax.

```
for (initialization; condition; final expression) {
    // code to run while the condition is true
}
```

This is a little more complicated than the `while` loop syntax:

- **Initialization** only happens once, when the code first kicks off. It's often used to set the initial value of the variable associated to the loop condition.
- The **condition** is evaluated once before the loop runs each time. If it's true, the code runs. If not, the code doesn't run.
- The **final expression** is evaluated after the loop runs each time. It's often used to update the value of the variable associated to the loop condition, as we saw in the previous example.

The loop counter

The variable used during initialization, condition, and the final expression of a loop is called a **counter** and often named `i`. It can be declared in the loop initialization to limit its scope to the loop body.

```
for (let i = 1; i <= 5; i++) {
    console.log(i);
}
// The i variable is not visible here
```

Common mistakes

Infinite `while` loop

The main risk with `while` loops is producing an **infinite loop**, meaning the condition is always true, and the code runs forever. This will crash your program! For example, let's say you forget a code line that increments the `number` variable.

```
let number = 1;
while (number <= 5) {
    console.log(number);
    // The number variable is not updated: the loop condition stays true fore\
ver
}
```

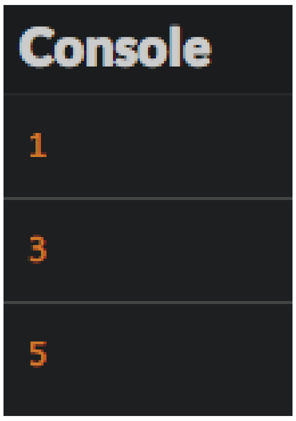
To protect yourself from infinite loops, you have to make sure the loop condition will eventually become false.

Manipulating a `for` loop counter

Imagine that you accidentally modify the loop counter in the loop body, just like in the following example.

```
for (let i = 1; i <= 5; i++) {
    console.log(i);
    i++; // The i variable is updated in the loop body
}
```

This program produces the following result.



```
Console
1
3
5
```

Execution result

Each time the loop runs, the counter variable is incremented *twice*: once in the body and once in the final expression after the loop runs. When you're using a `for` loop, you'll almost always want to omit anything to do with the counter inside the body of your loop. Just leave it in that first line!

Which loop should I use?

`For` loops are great because they include the notion of counting by default, avoiding the problem of infinite loops. However, it means you have to know how many times you want the loop to run as soon as you write your code. For situations where you don't already know how many times the code should run, `while` loops make sense. Here's a `while` loop use case in which a user is asked to type letters over and over until entering X:

```
let letter = "";
while (letter !== "X") {
    letter = prompt("Type a letter or X to exit:");
}
```

You can’t know how many times it’ll take for the user to enter `X`, so `while` is generally good for loops that depend on user interaction.

Ultimately, choosing which loop to use depends on context. All loops can be written with `while`, but if you know in advance how many times you want the loop to run, `for` is the best choice.

Coding time!

Yry to code each exercise twice, once with a `while` loop and the other with a `for`, to see for yourself which one is the most appropriate.

Carousel

Write a program that launches a carousel for 10 turns, showing the turn number each time.

When it’s done, improve it so that the number of turns is given by the user.

Parity

Check the following program that shows even number (divisibles by 2) betwen 1 and 10.

```
for (let i = 1; i <= 10; i++) {
    if (i % 2 === 0) {
        console.log(`${i} is even`);
    }
}
```

This program uses the modulo operator `%`, which calculates the remainder after division of one number by another. It’s often used to assess number parity.

```
console.log(10 % 2); // 0 because 10 = 5 * 2 + 0
console.log(11 % 2); // 1 because 11 = 5 * 2 + 1
console.log(18 % 3); // 0 because 18 = 3 * 6 + 0
console.log(19 % 3); // 1 because 19 = 3 * 6 + 1
console.log(20 % 3); // 2 because 20 = 3 * 6 + 2
```

Improve the program so that it also shows odd numbers. Improve it again so that the initial number is given by the user.

This program must show exactly 10 numbers including the first one, not 11!

Input validation

Write a program that asks the user for a number until it's less than or equal to `100`.

When it's done, improve it so that the number is between `50` and `100`.

Multiplication table

Write a program that asks the user for a number, then shows the multiplication table for this number.

When it's done, improve it to only accept numbers between `2` and `9` (use the previous exercise as a blueprint).

Neither yes nor no

Writea program that plays "neither yes, nor no" with the user: he enters a text until he types either "yes" or "no", ending the game.

FizzBuzz

Write a program that shows all numbers between `1` and `100` with the following exceptions:

- It shows `"Fizz"` instead if the number is divisible by `3`.
- It shows `"Buzz"` instead if the number is divisible by `5` and not by `3`.

When it's done, improve it so that it shows `"FizzBuzz"` instead of numbers divisible both by `3` and by `5`.

> This exercise has many, many solutions[1]. It's a job interview classic[2] that a significant number of candidates fail. Try your best!

[1]http://www.tomdalling.com/blog/software-design/fizzbuzz-in-too-much-detail/
[2]http://imranontech.com/2007/01/24/using-fizzbuzz-to-find-developers-who-grok-coding/

5. Write functions

In this chapter, you'll learn how to break down a program into subparts called functions.

TL;DR

- A **function** is a group of statements that performs a particular task. JavaScript functions are created using the `function` keyword.
- Written as a combinaison of several short and focused functions, a program will generally be easier to understand and more **modular** than a monolitic one.
- A **function call** triggers the execution of the function code. After it's done, execution resumes at the place where the call was made.
- Variables declared inside a function are limited in scope to the function body. They are called **local variables**.
- A `return` statement inside the function body defines the **return value** of the function. A function can accept zero, one or several **parameters** in order to work. For a particular call, supplied parameter values are called **arguments**.
- There are several ways to create a function in JavaScript. A first one is to use a **function declaration**.

```
// Function declaration
function myFunction(param1, param2, ...) {
    // Function code using param1, param2, ...
}

// Function call
myFunction(arg1, arg2, ...);
```

- Another way to create a function is to use a **function expression**. A function expression can be assigned to a variable because in JavaScript, a variable's value can be a function. Function expressions are often used to create **anonymous functions** (functions without a name).

```
// Anonymous function created with a function expression and assigned to a va\
riable
const myVar = function(param1, param2, ...) {
    // Function code using param1, param2, ...
}

// Function call
myVar(arg1, arg2, ...);
```

- A third way to create an anonymous function is the more recent **fat arrow syntax**.

```
// Fat arrow anonymous function assigned to a variable
const myVar = (param1, param2, ...)  => {
    // Function code using param1, param2, ...
}

// Function call
myVar(arg1, arg2, ...);
```

- No matter how it's created, each function should have a precise **purpose** and a well chosen **name** (often including an action verb). JavaScript offers a lots a **predefined functions** covering various needs.

Introduction: the role of functions

To understand why functions are important, check out our example from a previous chapter: the burrito algorithm :)

```
Begin
    Get out the rice cooker
    Fill it with rice
    Fill it with water
    Cook the rice
    Chop the vegetables
    Stir-fry the vegetables
    Taste-test the vegetables
        If the veggies are good
            Remove them from the stove
        If the veggies aren't good
            Add more pepper and spices
        If the veggies aren't cooked enough
            Keep stir-frying the veggies
    Heat the tortilla
    Add rice to tortilla
    Add vegetables to tortilla
    Roll tortilla
End
```

Here’s the same general idea, written in a different way.

```
Begin
    Cook rice
    Stir-fry vegetables
    Add fillings
    Roll together
End
```

The first version details all the individual actions that make up the cooking process. The second breaks down the recipe into **broader steps** and introduces concepts that could be re-used for other dishes as well like *cook*, *stir-fry*, *add* and *roll*.

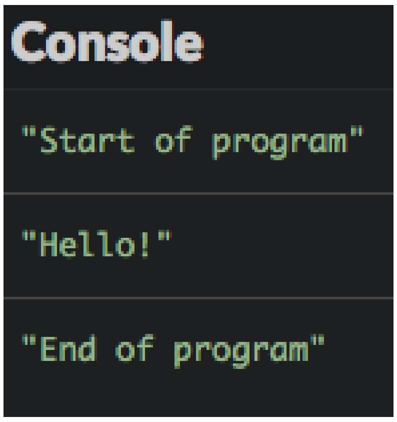
Our programs so far have mimicked the first example, but it’s time to start modularizing our code into sub-steps so we can re-use bits and pieces as needed. In JavaScript, these sub-steps are called **functions**!

Discovering functions

A **function** is a group of statements that performs a particular task.

Here’s a basic example of a function.

```
function sayHello() {
    console.log("Hello!");
}

console.log("Start of program");
sayHello();
console.log("End of program");
```



```
Console
"Start of program"
"Hello!"
"End of program"
```

Execution result

Let's study what just happened.

Declaring a function

Check out the first lines of the example above.

```
function sayHello() {
    console.log("Hello!");
}
```

This creates a function called `sayHello()`. It consists of only one statement that will make a message appear in the console: `"Hello!"`.

This is an example of a function **declaration**.

```
// Declare a function called myFunction
function myFunction() {
    // Function code
}
```

The declaration of a function is performed using the JavaScript keyword `function`, followed by the function name and a pair of parentheses. Statements that make up the function constitute the **body** of the function. These statements are enclosed in curly braces and indented.

Calling a function

Functions must be called in order to actually run. Here's the second part of our example program.

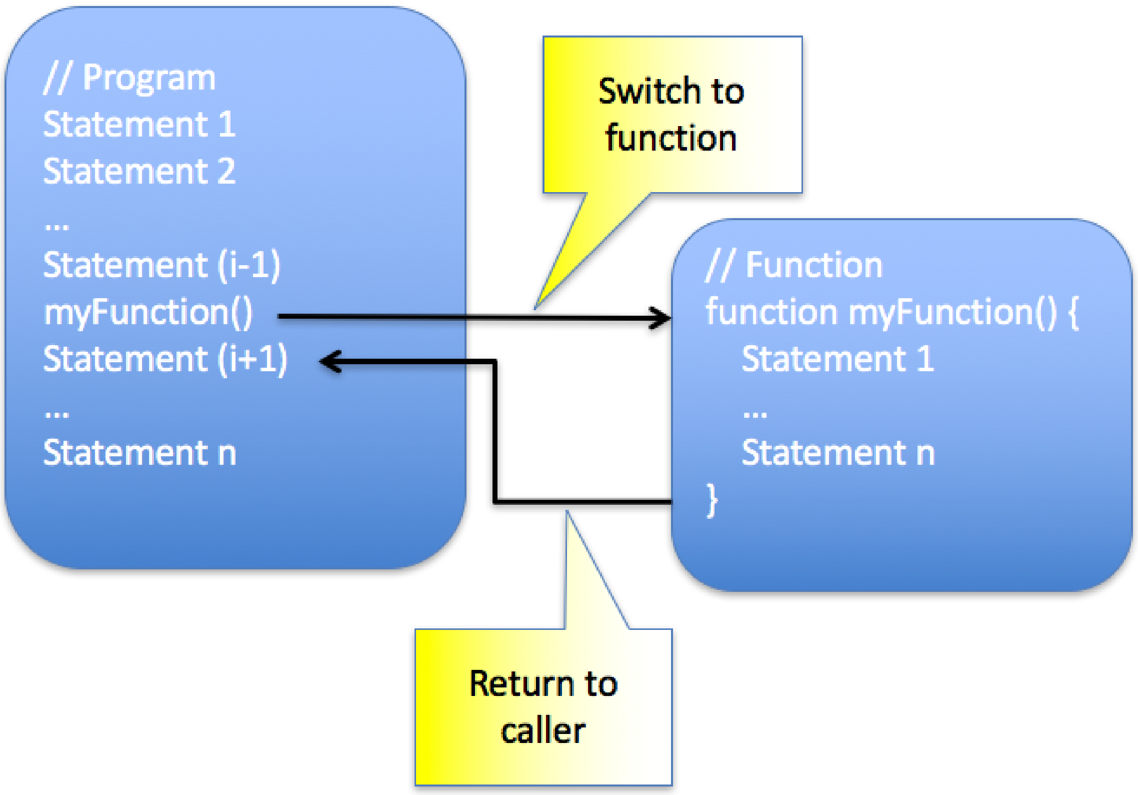
```
console.log("Start of program");
sayHello();
console.log("End of program");
```

The first and third statements explicitly display messages in the console. The second line makes a **call** to the function `sayHello()`.

You can call a function by writing the name of the function followed by a pair of parentheses.

```
// ...
myFunction(); // Call myFunction
// ...
```

Calling a function triggers the execution of actions listed therein (the code in its body). After it's done, execution resumes at the place where the call was made.



Function call mechanism

Usefulness of functions

A complex problem is generally more manageable when broken down in simpler subproblems. Computer programs are no exception to this rule. Writen as a combinaison of several short and focused functions, a program will be easier to understand and to update than a monolitic one. As an added bonus, some functions could be reused in other programs!

Creating functions can also be a solution to the problem of code duplication[1]: instead of being duplicated at several places, a piece of code is centralized in a function and called from anywhere when needed.

[1]https://en.wikipedia.org/wiki/Duplicate_code

Function contents

Return value

Here is a variation of our example program.

```
function sayHello() {
    return "Hello!";
}

console.log("Start of program");
const message = sayHello(); // Store the function return value in a variable
console.log(message);       // Show the return value
console.log("End of program");
```

Run this code, and you'll see the same result as before.

In this example, the body of the `sayHello()` function has changed: the statement `console.log("Hello!")` was replaced by `return "Hello!"`.

The keyword `return` indicates that the function will return a value, which is specified immediately after the keyword. This **return value** can be retrieved by the caller.

```
// Declare myFunction
function myFunction() {
    let returnValue;
    // Calculate return value
    // returnValue = ...
    return returnValue;
}

// Get return value from myFunction
const result = myFunction();
// ...
```

This return value can be any type (number, string, etc). However, a function can return only one value.

Retrieving a function's return value is not mandatory, but in that case the return value is "lost".

If you try to retrieve the return value of a function that does not actually have one, we get the JavaScript value `undefined`.

```
function myFunction() {
    // ...
    // No return value
}

const result = myFunction();
console.log(result); // undefined
```

A function stops running immediately after the `return` statement is executed. Any further statements would never be run.

Let's simplify our example a bit by getting rid of the variable that stores the function's return value.

```
function sayHello() {
    return "Hello!";
}

console.log(sayHello()); // "Hello!"
```

The return value of the `sayHello()` function is directly output through the `console.log()` command.

Local variables

You can declare variables inside a function, as in the example below.

```
function sayHello() {
    const message = "Hello!";
    return message;
}

console.log(sayHello()); // "Hello!"
```

The function `sayHello()` declares a variable named `message` and returns its value.

The variables declared in the body of a function are called **local variables**. Their **scope** is limited to the function body (hence their name). If you try to use it outside the function, you won't be able to!

```
function sayHello() {
    const message = "Hello!";
    return message;
}

console.log(sayHello()); // "Hello!"
console.log(message);    // Error: the message variable is not visible here
```

Each function call will redeclare the function's local variables, making the calls perfectly independent from one another.

Not being able to use local variables outside the functions in which they are declared may seem like a limitation. Actually, it's a good thing! This means functions can be designed as autonomous and reusable. Moreover, this prevents **naming conflicts**: variables declared in different functions may have the same name.

Parameter passing

A **parameter** is an information that the function needs in order to work. The function parameters are defined in parentheses immediately following the name of the function. You can then use the parameter value in the body of the function.

You supply the parameter value when calling the function. This value is called an **argument**.

Let's edit the above example to add a personalized greeting:

```
function sayHello(name) {
    const message = `Hello, ${name}!`;
    return message;
}

console.log(sayHello("Baptiste")); // "Hello, Baptiste!"
console.log(sayHello("Thomas"));   // "Hello, Thomas!"
```

The declaration of the `sayHello()` function now contains a parameter called `name`.

In this example, the first call to `sayHello()` is done with the argument `"Baptiste"` and the second one with the argument `"Thomas"`. In the first call, the value of the `name` parameter is `"Baptiste"`, and `"Thomas"` in the second.

Here's the general syntax of a function declaration with parameters. The number of parameters is not limited, but more than 3 or 4 is rarely useful.

```
// Declare a function myFunction with parameters
function myFunction(param1, param2, ...) {
    // Statements using param1, param2, ...
}

// Function call
// param1 value is set to arg1, param2 to arg2, ...
myFunction(arg1, arg2, ...);
```

Just like with local variables, parameter scope is limited to the function body. Thus, an external variable used as an argument in a function call may have the same name as a function parameter. The following example is perfectly valid.

```
function sayHello(name) {
    // Here, "name" is the function parameter
    const message = `Hello, ${name}!`;
    return message;
}

// Here, "name" is a variable used as an argument
let name = "Baptiste";
console.log(sayHello(name)); // "Hello, Baptiste!"
name = "Thomas";
console.log(sayHello(name)); // "Hello, Thomas!"
```

When calling a function, respecting the number and order of parameters is paramount! Check out the following example.

```
function presentation(name, age) {
    console.log(`Your name is ${name} and you're ${age} years old`);
}

presentation("Garance", 9); // "Your name is Garance and you're 9 years old"
presentation(5, "Prosper"); // "Your name is 5 and you're Prosper years old"
```

The second call arguments are given in reverse order, so `name` gets the value `5` and `age` gets `"Prosper"` for that call.

Anonymous functions

Declaration is not the only way to create functions in JavaScript. Check out this example.

```
const hello = function(name) {
    const message = `Hello, ${name}!`;
    return message;
}

console.log(hello("Richard")); // "Hello, Richard!"
```

In this example, the function is assigned to the `hello` variable. The value of this variable is a function. We call the function using that variable. This is an example of a **function expression**. A function expression defines a function as part of a larger expression, typically a variable assignment.

The function created in this example has no name: it is **anonymous**. As you'll soon discover, anonymous functions are heavily used in JavaScript.

Here's how to create an anonymous function and assign it to a variable.

```
// Assignment of an anonymous function to the myVar variable
const myVar = function(param1, param2, ...) {
    // Statements using param1, param2, ...
}

// Anonymous function call
// param1 value is set to arg1, param2 to arg2, ...
myVar(arg1, arg2, ...);
```

Recent language evolutions have introduced a more concise way to create anonymous functions:

```
const hello = (name) => {
    const message = `Hello, ${name}!`;
    return message;
}

console.log(hello("William")); // "Hello, William!"
```

Functions created this way are called **fat arrow functions**.

```
// Assignment of an anonymous function to the myVar variable
const myVar = (param1, param2, ...)  => {
    // Statements using param1, param2, ...
}

// Anonymous function call
// param1 value is set to arg1, param2 to arg2, ...
myVar(arg1, arg2, ...);
```

Fat arrow function syntax can be further simplified in some particular cases:

- When there's only one statement in the function body, everything can be written on the same line without curly braces. The `return` statement is omitted and implicit.
- When the function accepts only one parameter, parentheses around it can be omitted.

```
// Minimalist to the max
const hello = name => `Hello, ${name}!`;

console.log(hello("Kate")); // "Hello, Kate!"
```

Functions are a core part of the JavaScript toolset. You'll use them non-stop in your programs.

Guidelines for programming with functions

Creating functions wisely

Functions can include everything you can use in a regular program: variables, conditionals, loops, etc. Functions can call one another, giving the programmer an enormous amount of freedom for building programs.

However, not everything deserves to be in its own function. it's better to write short and focused ones, in order to limit dependencies and improve program understanding.

Leveraging JavaScript predefined functions

We have already used several predefined JavaScript functions like `prompt()` and `alert()`. They are many others in the language specification. Get to know them instead of reinventing the wheel!

Here' an example demonstrating two of the JavaScript mathematical functions.

```
console.log(Math.min(4.5, 5)); // 4.5
console.log(Math.min(19, 9));  // 9
console.log(Math.min(1, 1));   // 1
console.log(Math.random());    // A random number between 0 and 1
```

The function `Math.min()` return the minimum number among its arguments. The function `Math.random()` generates a random number betwen 0 and 1.

This book will introduce many other JavaScript functions.

Limiting function complexity

A function body must be kept simple, or otherwise split into several sub-functions. A a rule of thumb, 30 lines of code should be a max for non-specific cases.

Naming functions and parameters well

Function naming is just as important as variable naming. You should choose names that express clearly the function purpose and follow a naming convention like camelCase[2].

A popular practice is to include in the name an **action verb** like *calculate*, *show*, *find*, etc.

If you have difficulties coming up with a right name for a function, then maybe its purpose is not that clear and you should ask yourself if this function deserves to exist.

Coding time!

Improved hello

Complete the following program so that it asks the user for his first and last names, then show the result of the `sayHello()` function.

```
// Say hello to the user
function sayHello(firstName, lastName) {
    const message = `Hello, ${firstName} ${lastName}!`;
    return message;
}

// TODO: ask user for first and last name
// TODO: call sayHello() and show its result
```

Number squaring

Complete the following program so that the `square1()` and `square2()` functions work properly.

[2]https://en.wikipedia.org/wiki/Camel_case

```
// Square the given number x
function square1(x) {
  // TODO: complete the function code
}

// Square the given number x
const square2 = x => {
  // TODO: complete the function code
}

console.log(square1(0)); // Must show 0
console.log(square1(2)); // Must show 4
console.log(square1(5)); // Must show 25

console.log(square2(0)); // Must show 0
console.log(square2(2)); // Must show 4
console.log(square2(5)); // Must show 25
```

When it's done, update the program so that it shows the square of every number between 0 en 10.

> Writing 10 dumb calls to `square()` is forbidden! You know how to repeat statements, don't you ? ;)

Minimum of two numbers

Let's pretend the JavaScript `Math.min()` function doesn't exist. Complete the following program so that the `min()` function returns the minimum of its two received numbers.

```
// TODO: write the min() function

console.log(min(4.5, 5)); // Must show 4.5
console.log(min(19, 9));  // Must show 9
console.log(min(1, 1));   // Must show 1
```

Calculator

Complete the following program so that it offers the four basic arithmetical operations: addition, substration, multiplication and division. You can use either a function declaration or a function expression.

```
// TODO: complete program

console.log(calculate(4, "+", 6));  // Must show 10
console.log(calculate(4, "-", 6));  // Must show -2
console.log(calculate(2, "*", 0));  // Must show 0
console.log(calculate(12, "/", 0)); // Must show Infinity
```

Circumference and area of a circle

Write a program containing two functions to calculate the circumference and area of a square defined by its radius. Test it using user input.

> Circumference and area calculation formulas should be part of your school years memories… Or a Google click away :)

> The value of number π (Pi) is obtained with `Math.PI` in JavaScript.

6. Create your first objects

This chapter will introduce objects and the way they are created and used in JavaScript.

TL;DR

- A JavaScript **object** is an entity that has properties. Each property is a key/value pair. The key is the property name.
- The value of a property can be an information (number, string, etc) or a function. In that case, the property is called a **method**.
- A JavaScript **object literal** is created by simply setting its properties within a pair of curly braces.

```
const myObject = {
    property1: value1,
    property2: value2,
    // ... ,
    method1(/* ... */) {
        // ...
    },
    method2(/* ... */) {
        // ...
    },
    // ...
};

myObject.property1 = newValue;   // Set the new value of property1 for myObje\
ct
console.log(myObject.property1); // Show the value of property1 for myObject
myObject.method1(...);           // Call method1 on myObject
```

- Inside a method, the `this` keyword represents the object on which the method is called.
- The JavaScript language predefines many useful objects like `console` or `Math`.

Introduction

What's an object?

Think about objects in the non-programming sense, like a pen. A pen can have different ink colors, be manufactured by different people, have a different tip, and many other properties.

Similarly, an **object** in programming is an **entity that has properties****. Each property defines a characteristic of the object. A property can be a information associated with the object (the color of the pen) or an action (the pen's ability to write).

What does this have to do with code?

Object-oriented programming (OOP for short) is a way to write programs using objects. When using OOP, you write, create, and modify objects, and the objects make up your program.

OOP changes the way a program is written and organized. So far, you've been writing function-based code, sometimes called procedural programming[1]. Now let's discover how to write object-oriented code.

JavaScript and objects

Like many other languages, JavaScript supports programming with objects. It provides a number of predefined objects while also letting you create your own.

Creating an object

Here is the JavaScript representation of a blue Bic ballpoint pen.

```
const pen = {
    type: "ballpoint",
    color: "blue",
    brand: "Bic"
};
```

A JavaScript object can be created by simply setting its properties within a pair of curly braces: `{...}`. Each property is a key/value pair. This is called an **object literal**.

> The semicolon `;` after the closing brace is optional, but it's safer to add it anyway.

The above code defines a variable named `pen` whose value is an object: you can therefore say `pen` is an object. This object has three properties: `type`, `color` and `brand`. Each property has a name and a value and is separated by a comma `,` (except the last one).

Accessing an object's properties

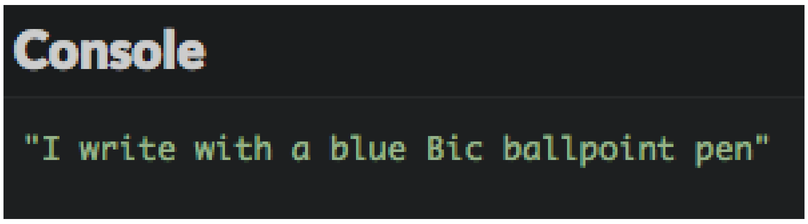
After creating an object, you can access the value of its properties using **dot notation** such as `myObject.myProperty`.

[1] https://en.wikipedia.org/wiki/Procedural_programming

```
const pen = {
    type: "ballpoint",
    color: "blue",
    brand: "Bic"
};

console.log(pen.type);  // "ballpoint"
console.log(pen.color); // "blue"
console.log(pen.brand); // "Bic"
```

Accessing an object's property is an **expression** that produces a value. Such an expression can be included in more complex ones. For example, here's how to show our pen properties in one statement.

```
const pen = {
    type: "ballpoint",
    color: "blue",
    brand: "Bic"
};

console.log(`I write with a ${pen.color} ${pen.brand} ${pen.type} pen`);
```



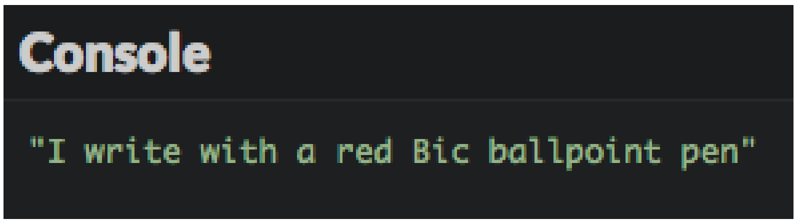
```
"I write with a blue Bic ballpoint pen"
```

Execution result

Modifying an object

Once an object is created, you can change the value of its properties with the syntax `myObject.myProperty = newValue`.

```
const pen = {
    type: "ballpoint",
    color: "blue",
    brand: "Bic"
};

pen.color = "red"; // Modify the pen color property

console.log(`I write with a ${pen.color} ${pen.brand} ${pen.type} pen`);
```

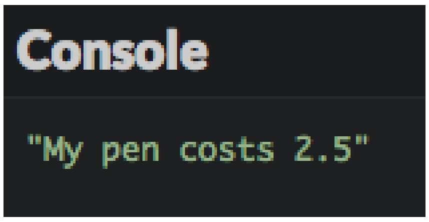


```
Console
"I write with a red Bic ballpoint pen"
```

Execution result

JavaScript even offers the ability to dynamically add new properties to an already created object.

```
const pen = {
    type: "ballpoint",
    color: "blue",
    brand: "Bic"
};

pen.price = "2.5"; // Set the pen price property

console.log(`My pen costs ${pen.price}`);
```



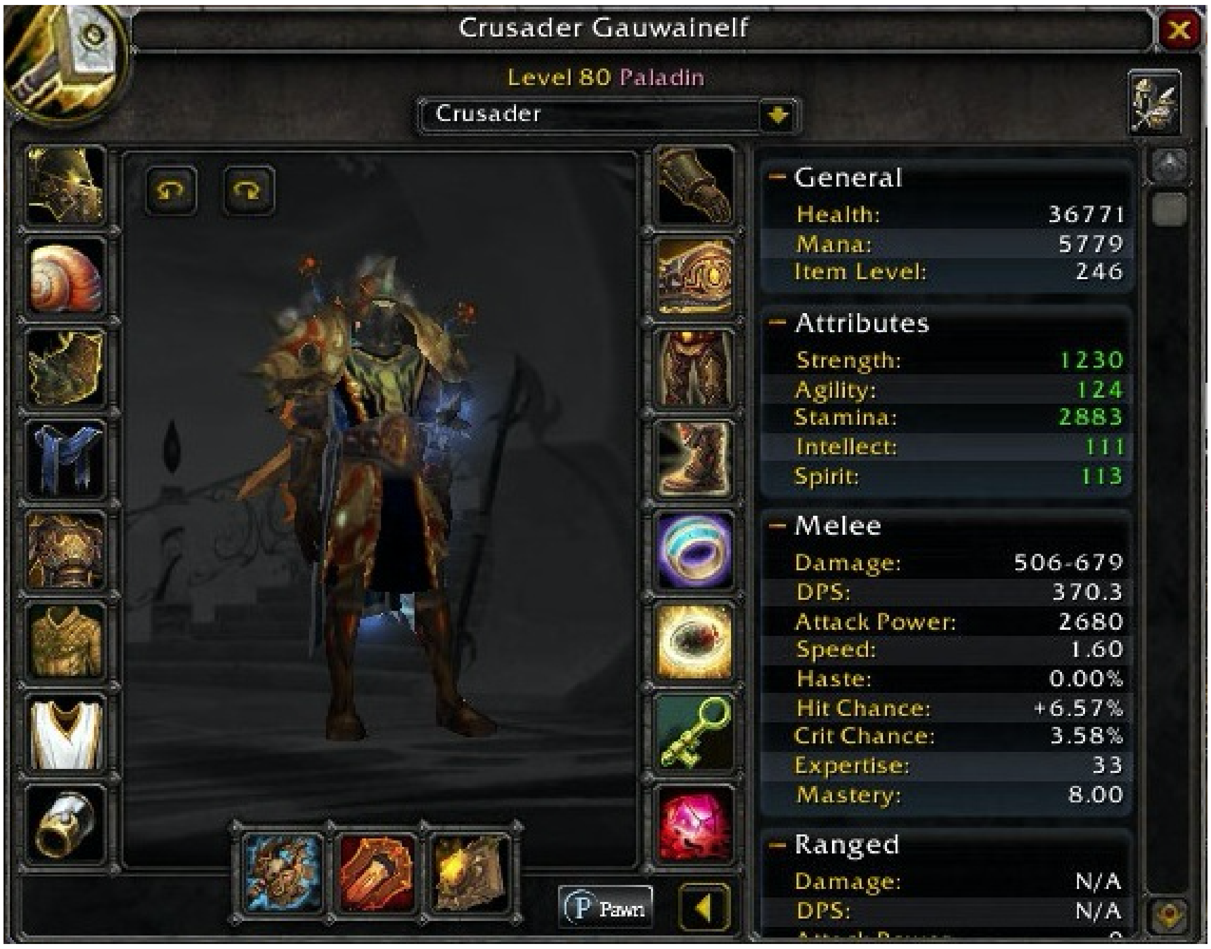
```
Console
"My pen costs 2.5"
```

Execution result

Programming with objects

Many books and courses teach object-oriented programming through examples involving animals, cars or bank accounts. Let's try something cooler and create a mini-role playing game (RPG) using objects.

In a role-playing game, each character is defined by many attributes like strength, stamina or intelligence. Here's the character screen of a very popular online RPG.



General

Health:	36771
Mana:	5779
Item Level:	246

Attributes

Strength:	1230
Agility:	124
Stamina:	2883
Intellect:	111
Spirit:	113

Melee

Damage:	506-679
DPS:	370.3
Attack Power:	2680
Speed:	1.60
Haste:	0.00%
Hit Chance:	+6.57%
Crit Chance:	3.58%
Expertise:	33
Mastery:	8.00

Ranged

Damage:	N/A
DPS:	N/A

No, it's not mine!

In our simpler example, a character will have three attributes:

- his name,
- his health (number of life points),
- his strength.

A naive example

Let me introduce you to Aurora, our first RPG character.

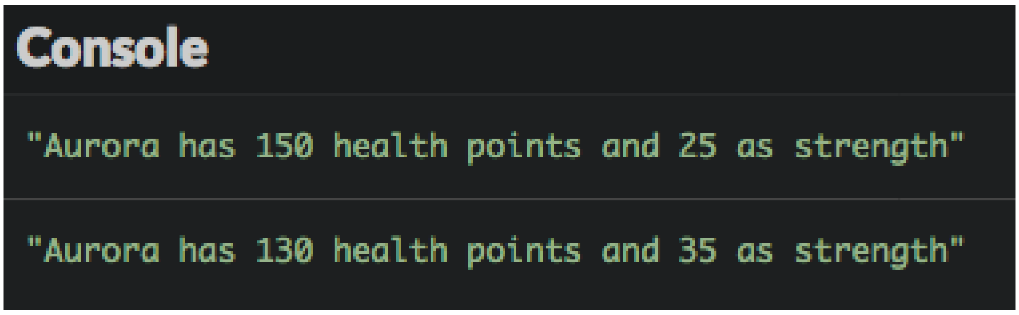
```
const aurora = {
    name: "Aurora",
    health: 150,
    strength: 25
};
```

The `aurora` object has three properties: `name`, `health` and `strength`.

As you can see, you can assign numbers, strings, and even other objects to properties!

Aurora is about to start a series of great adventures, some of which will update her attributes. Check out the following example.

```
const aurora = {
    name: "Aurora",
    health: 150,
    strength: 25
};

console.log(`${aurora.name} has ${aurora.health} health points and ${aurora.s\
trength} as strength`);

// Aurora is harmed by an arrow
aurora.health = aurora.health - 20;

// Aurora equips a strength necklace
aurora.strength = aurora.strength + 10;

console.log(`${aurora.name} has ${aurora.health} health points and ${aurora.s\
trength} as strength`);
```



```
Console
"Aurora has 150 health points and 25 as strength"
"Aurora has 130 health points and 35 as strength"
```

Execution result

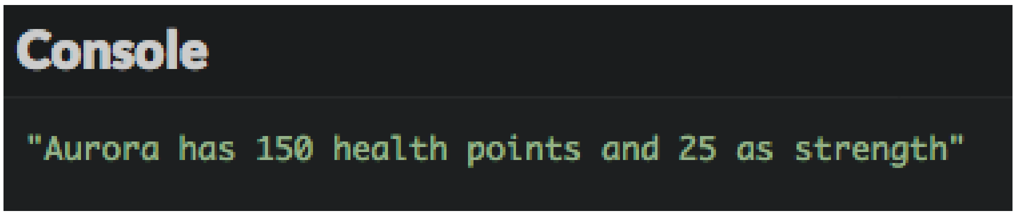
Introducing methods

In the above code, we had to write lengthy `console.log` statements each time to show our character state. There's a cleaner way to accomplish this.

Adding a method to an object

Observe the following example.

```
const aurora = {
    name: "Aurora",
    health: 150,
    strength: 25
};

// Return the character description
function describe(character) {
    return `${character.name} has ${character.health} health points and ${cha\
racter.strength} as strength`;
}

console.log(describe(aurora));
```



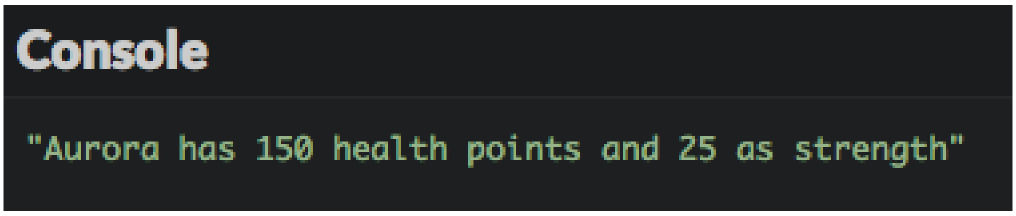
```
Console
"Aurora has 150 health points and 25 as strength"
```

Execution result

The `describe()` function takes an object as a parameter. It accesses that object's properties to create a description string.

Now for an alternative approach: creating a `describe()` property *inside* the object.

```
const aurora = {
    name: "Aurora",
    health: 150,
    strength: 25,

    // Return the character description
    describe() {
        return `${this.name} has ${this.health} health points and ${this.stre\
ngth} as strength`;
    }
};

console.log(aurora.describe());
```



```
Console
"Aurora has 150 health points and 25 as strength"
```

Execution result

Now our object has a new property available to it: `describe()`. The value of this property is a function that returns a textual description of the object. The execution result is exactly the same as before.

An object property whose value is a function is called a **method**. Methods are used to define **actions** for an object. A method add some **behavior** to an object.

Calling a method on an object

Let's look at the last line of our previous example.

```
console.log(aurora.describe());
```

To show the character description, we use the `aurora.describe()` expression instead of `describe(aurora)`. It makes an *essential* difference:

- `describe(aurora)` calls the `describe()` function with the `aurora` object as an argument. The function is external to the object. This is an example of procedural programming.
- `aurora.describe()` calls the `describe()` function on the `aurora` object. The function is one of the object's properties: it is a method. This is an example of object-oriented programming.

To call a method named `myMethod()` on an object `myObject`, the syntax is `myObject.myMethod()`.

Remember the parentheses, even if empty, when calling a method!

The `this` keyword

Now look closely at the body of the `describe()` method on our object.

```
const aurora = {
    name: "Aurora",
    health: 150,
    strength: 25,

    // Return the character description
    describe() {
        return `${this.name} has ${this.health} health points and ${this.stre\
ngth} as strength`;
    }
};
```

You see a new keyword: `this`. This is automatically set by JavaScript inside a method and represents **the object on which the method was called**.

The `describe()` method doesn't take any parameters. It uses `this` to access the properties of the object on which it is called.

JavaScript predefined objects

The JavaScript language has many predefined objects serving various purposes. We have already encountered some of them:

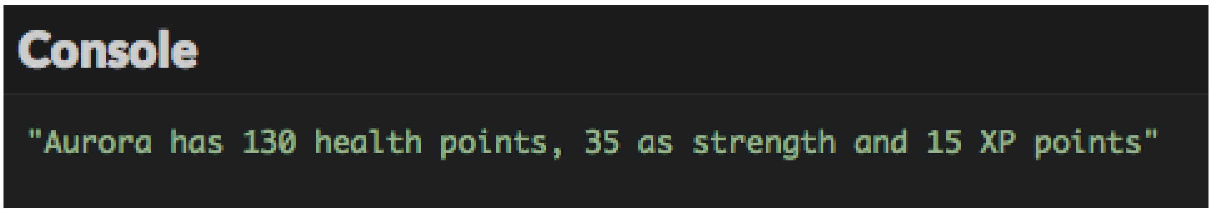
- The `console` object gives access to the environment console. `console.log()` is actually a method call.
- The `Math` object contains many mathematical properties. For example, `Math.PI` returns an approximate value of the number π (Pi) and the `Math.random()` function returns a random number between `0` and `1`.

Coding time!

Adding character experience

Improve our example RPG program to add an experience property named `xp` to the character. Its initial value is `0`. Experience must appear in character description.

```
// TODO: create the character object here

// Aurora is harmed by an arrow
aurora.health = aurora.health - 20;

// Aurora equips a strength necklace
aurora.strength = aurora.strength + 10;

// Aurora learn a new skill
aurora.xp = aurora.xp + 15;

console.log(aurora.describe());
```



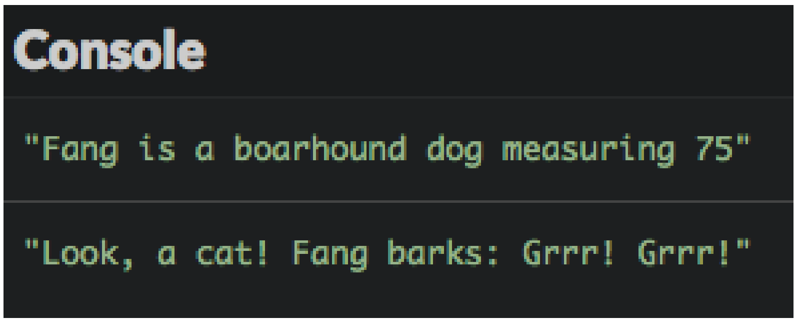
```
Console
"Aurora has 130 health points, 35 as strength and 15 XP points"
```

Execution result

Modeling a dog

Complete the following program to add the `dog` object definition.

```
// TODO: create the dog object here

console.log(`${dog.name} is a ${dog.species} dog measuring ${dog.size}`);
console.log(`Look, a cat! ${dog.name} barks: ${dog.bark()}`);
```



```
Console
"Fang is a boarhound dog measuring 75"
"Look, a cat! Fang barks: Grrr! Grrr!"
```

Execution result

Modeling a circle

Complete the following program to add the `circle` object definition. Its radius value is input by the user.

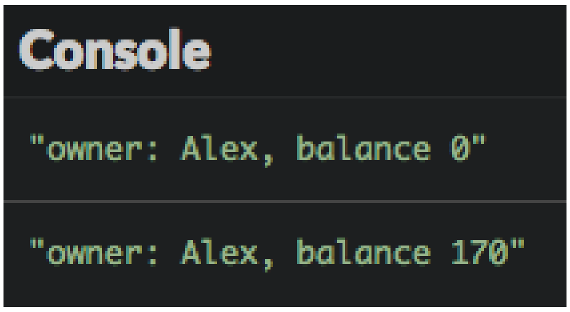
```
const r = Number(prompt("Enter the circle radius:"));

// TODO: create the circle object here

console.log(`Its circumference is ${circle.circumference()}`);
console.log(`Its area is ${circle.area()}`);
```

Modeling a bank account

Write a program that creates an `account` object with the following characteristics:

- A `name` property set to "Alex".
- A `balance` property set to `0`.
- A `credit` method adding the value passed as an argument to the account balance.
- A `describe` method returning the account description.

Use this object to show its description, crediting 250, debiting 80, then show its description again.



```
Console
"owner: Alex, balance 0"
"owner: Alex, balance 170"
```

Execution result

7. Store data in arrays

This chapter will introduce you to arrays[1], a type of variable used in many computer programs to store data.

TL;DR

- An **array** represents a set of elements. A JavaScript array is an object that has special properties, like `length` to access its size (number of elements).
- You can think of an array as a set of boxes, each storing a specific value and associated with a number called its **index**. The first element of an array will be index number `0` - not 1.
- You can access a particular element by passing its index within **square brackets** `[]`.
- To iterate over an array (browsing it element by element), you can use the `for` loop, the `forEach()` method or the newer `for-of` loop.

```
for (let i = 0; i < myArray.length; i++) {
    // Use myArray[i] to access each array element one by one
}

myArray.forEach(myElement => {
    // Use myElement to access each array element one by one
});

for (const myElement of myArray) {
    // Use myElement to access each array element one by one
}
```

- The `push()` method adds an element ad the end of an array.
- The `pop()` and `splice()` are used to remove elements from the array.

Introduction to arrays

Imagine you want to create a list of all the movies you've seen this year.

One solution would be to create several variables:

[1] https://en.wikipedia.org/wiki/Array_data_type

```
var movie1 = "The Wolf of Wall Street";
var movie2 = "Zootopia";
var movie3 = "Babysitting";
var movie4 = "Metropolis";
var movie5 = "Gone with the Wind";
// ...
```

If you're a movie buff, you may find yourself with too many variables in your program. The worst part is that these variables are completely independent from one another.

Another possibility is to group the movies in an object.

```
const movies = {
    movie1: "The Wolf of Wall Street",
    movie2: "Zootopia",
    movie3: "Babysitting",
    movie4: "Metropolis",
    movie5: "Gone with the Wind",
    // ...
};
```

This time, the data is centralized in the object `movies`. The names of its properties (`movie1`, `movie2`, `movie3`...) are, however, unnecessary and repetitive.

You need a solution to store items together without naming them individually!

Luckily, there is one: use an array. An **array** is a data type that can store a set of elements.

Manipulating arrays in JavaScript

In JavaScript, an array is an object that has special properties.

Creating an array

Here's how to create our list of movies in the form of an array.

```
const movies = ["The Wolf of Wall Street", "Zootopia", "Babysitting", "Metrop\
lis", "Gone with the Wind"];
```

An array is created with a pair of square brackets `[]`. Everything within the brackets makes up the array.

You can store different types of elements within an array, including strings, numbers, booleans and even objects.

```
const elements = ["Hello", 7, { message: "Hi mom" }, true];
```

Since an array may contain multiple elements, it's good to name the array plurally (for example, `movies`).

Obtaining an array's size

The number of elements stored in an array is called its **size**. Here's how to access it.

```
const movies = ["The Wolf of Wall Street", "Zootopia", "Babysitting", "Metrop\
lis", "Gone with the Wind"];
console.log(movies.length); // 5
```

You access the size of an array via its `length` property, using the dot notation.

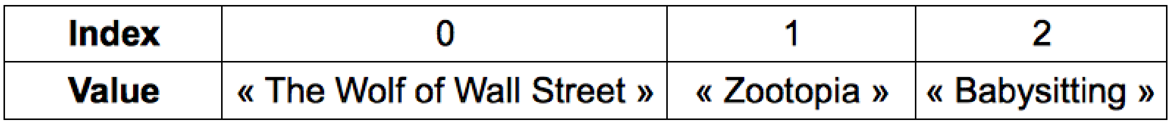
Of course, this `length` property returns 0 in case of an empty array.

```
const emptyArray = [];           // Create an empty array
console.log(emptyArray).length); // 0
```

Access an element in an array

Each item in an array is identified by a number called its **index** - an integer pointer that identifies an element of the array. We can think of an array as a set of boxes, each storing a specific value and associated with an index. Here's the trick: the first element of an array will be index number 0 - not 1. The second element will be index number 1, and so on. The index of the last array element would be the array's size minus 1.

Here is how you might represent the `movies` array:



Index	0	1	2
Value	« The Wolf of Wall Street »	« Zootopia »	« Babysitting »

Movies array representation

You can access a particular element by passing its index within **square brackets** `[]`:

```
const movies = ["The Wolf of Wall Street", "Zootopia", "Babysitting", "Metrop\
lis", "Gone with the Wind"];
console.log(movies[0]); // "The Wolf of Wall Street"
console.log(movies[1]); // "Zootopia"
console.log(movies[2]); // "Babysitting"
console.log(movies[3]); // "Metroplis"
console.log(movies[4]); // "Gone with the Wind"
```

Using an invalid index to access a JavaScript array element returns the value `undefined`.

```
const movies = ["The Wolf of Wall Street", "Zootopia", "Babysitting", "Metrop\
lis", "Gone with the Wind"];
console.log(movies[5]); // undefined: last element is at index 4
```

Iterating over an array

There are several ways to browse an array element by element.

The first is to use a `for` loop as discussed previously.

```
const movies = ["The Wolf of Wall Street", "Zootopia", "Babysitting", "Metrop\
lis", "Gone with the Wind"];
for (let i = 0; i < movies.length; i++) {
    console.log(movies[i]);
}
```

The `for` loop runs through each element in the array starting with index 0 all the way up to the length of the array minus 1, which will be its last element.

Another way is to call the `forEach()` method on the array. It takes as a paramater a **function** that will be applied to each array element.

Here's the previous example, rewritten with this method and a fat arrow function.

```
const movies = ["The Wolf of Wall Street", "Zootopia", "Babysitting", "Metrop\
lis", "Gone with the Wind"];
movies.forEach(movie => {
    console.log(movie);
});
```

During execution, each array element is passed as a parameter (named `movie` in this example) to the anonymous function associated to `forEach()`.

Lastly, you can use the `for-of` loop, a special kind of loop dealing with iterable objects[2] like arrays. Here's its syntax.

[2]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Iteration_protocols#iterable

```
for (const myElement of myArray) {
    // Use myElement to access each array element one by one
}
```

Check out the previous example written with a `for-of` loop.

```
const movies = ["The Wolf of Wall Street", "Zootopia", "Babysitting", "Metrop\
lis", "Gone with the Wind"];
for (const movie of movies) {
    console.log(movie);
}
```

Updating an array’s content

Adding an element to an array

Don’t lie to me: you’ve just watched Ghostbusters *yet another time*. Let’s add it to the list. Here’s how you’d do so.

```
const movies = ["The Wolf of Wall Street", "Zootopia", "Babysitting", "Metrop\
lis", "Gone with the Wind"];
movies.push("Ghostbusters");
console.log(movies[5]); // "Ghostbusters"
```

You add a new item to an array with the `push()` method. The new element to be added is passed as a parameter to the method. It is inserted at the end of the array.

Removing an element from an array

You can remove the last element of an array using the `pop()` method.

```
const movies = ["The Wolf of Wall Street", "Zootopia", "Babysitting", "Metrop\
lis", "Gone with the Wind"];
movies.pop();                // Remove the last array element
console.log(movies.length); // 4
console.log(movies[4]);     // undefined
```

Alternatively, you can use the `splice()` method with two parameters: the first one is the index from which to begin removing, and the second one is the number of elements to remove.

```
const movies = ["The Wolf of Wall Street", "Zootopia", "Babysitting", "Metrop\
lis", "Gone with the Wind"];
movies.splice(0, 1);        // Remove 1 element starting at index 0
console.log(movies.length); // 4
console.log(movies[0]);     // "Zootopia"
console.log(movies[1]);     // "Babysitting"
```

More on Arrays

You can do many wonderful things with arrays. Here are some links that may help you deepen your understanding of arrays:

- Mozilla Developer Network Array - JavaScript[3]
- W3 Schools JavaScript Arrays[4]
- W3 Schools JavaScript Array Methods[5]
- W3 Schools JavaScript Array Reference[6]

Coding time!

Create all these programs in a generic fashion: the program output should reflect any update in the array's content.

Musketeers

Write a program that:

- Creates an array named `musketeers` containing values "Athos", "Porthos" and "Aramis".
- Shows each array element using a `for` loop.
- Adds the "D'Artagnan" value to the array.
- Shows each array element using the `forEach()` method.
- Remove poor Aramis.
- Shows each array element using a `for-of` loop.

Sum of values

Write a program that creates the following array, then calculates and shows the sum of its values (`42` in that case).

[3]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array
[4]https://www.w3schools.com/js/js_arrays.asp
[5]https://www.w3schools.com/js/js_array_methods.asp
[6]https://www.w3schools.com/jsref/jsref_obj_array.asp

```
const values = [3, 11, 7, 2, 9, 10];
```

Array maximum

Write a program that creates the following array, then calculates and shows its maximum value.

```
const values = [3, 11, 7, 2, 9, 10];
```

List of words

Write a program that asks the user for a word until he types `"stop"`. The program then shows each of these words, except `"stop"`.

8. Work with strings

A lot of code you write will involve modifying chains of text characters - or strings[1]. Let's look at how!

TL;DR

- Although string values are primitive JavaScript types, some **properties** and **methods** may be applied to them just as if they were objects.
- The `length` property returns the number of characters of the string.
- JavaScript string are **immutable**[2]: once created, a string value never changes. String methods never affect the initial value and always return a new string.
- The `toLowerCase()` and `toUpperCase()` methods respectively convert strings to lower and upper case.
- String values may be compared using the === operator, which is case sensitive.
- A string may be seen as an **array of characters** identified by their **index**. The index of the first character is `0` (not `1`).
- You may iterate over a string using either a `for` or the newer `for-of` loop.
- Searching for values inside a string is possible with the `indexOf()`, `startsWith()` and `endsWith()` methods.
- The `split()` method breaks a string into subparts delimited by a separator.

String recap

Let's recapitulate what we already know about strings:

- A string value represents text.
- In JavaScript, a string is defined by placing text within single quotes (`'I am a string'`) or double quotes (`"I am a string"`).
- You may use special characters within a string by prefacing them with `\` ("backslash") followed by another character. For example, use `\n` to add a linebreak.
- The + operator concatenates (combines or adds) two or more strings.

Beyond these basic uses, strings have even more versatilty.

Obtaining string length

To obtain the **length** of a string (the number of characters it contains), add `.length` to it. The length will be returned as an integer.

[1]https://en.wikipedia.org/wiki/String_(computer_science)
[2]https://en.wikipedia.org/wiki/Immutable_object

```
console.log("ABC".length); // 3
const m = "I am a string";
const l = m.length;
console.log(l); // 13
```

Although string values are primitive JavaScript types, some properties and methods can be applied to them just as if they were objects by using the **dot notation**. `length` is one of those properties.

Converting string case

You may convert a string's text to **lowercase** by calling the `toLowerCase()` method. Alternatively, you may do the same with `toUpperCase()` to convert a string to uppercase.

```
const originalWord = "Bora-Bora";

const lowercaseWord = originalWord.toLowerCase();
console.log(lowercaseWord); // "bora-bora"

const uppercaseWord = originalWord.toUpperCase();
console.log(uppercaseWord); // "BORA-BORA"
```

`toLowerCase()` and `toUpperCase()` are two string methods. Like every string method, both have no affect on the initial value and return a new string.

It's important to understand that once created, a string value never change: strings are **immutable** in JavaScript.

Comparing two strings

You may compare two strings with the === operator. The operation returns a boolean value: `true` if the strings are equal, `false` if not.

```
const word = "koala";
console.log(word === "koala");    // true
console.log(word === "kangaroo"); // false
```

String comparison is case sensitive. Do pay attention to your lower and uppercase letters!

```
console.log("Qwerty" === "qwerty");               // false
console.log("Qwerty".toLowerCase() === "qwerty"); // true
```

Strings as sets of characters

Identifying a particular character

You may think of a string as an array of characters. Each character is identified by a number called an index, just as it does for an array. The same golden rules apply:

- The index of the first character in a string is 0, not 1.
- The highest index number is the string's length minus 1.

Accessing a particular character

You know how to identify a character by its index. To access it, you use the **brackets notation** `[]` with the character index placed between the brackets.

Trying to access a string character beyond the string length produces an `undefined` result.

```
const sport = "basketball";
console.log(sport[0]);  // first "b"
console.log(sport[6]);  // second "b"
console.log(sport[10]); // undefined: last character is at index 9
```

Iterating over a string

Now what if you want to access all string characters one-by-one? You could access each letter individually, as seen above:

```
const name = "Sarah"; // 5 characters
console.log(name[0]); // "S"
console.log(name[1]); // "a"
console.log(name[2]); // "r"
console.log(name[3]); // "a"
console.log(name[4]); // "h"
```

This is impractical if your string contains more than a few characters. You need a better solution to *repeat* access to characters. Does the word "repeat" bring to mind a former concept? Loops, of course!

You may write a **loop** to access each character of a string. Generally speaking, a `for` loop is a better choice than a `while` loop, since we know here that the loop will need to run for each character in the string.

```
for (let i = 0; i < myString.length; i++) {
    // Use myString[i] to access each character one by one
}
```

The loop counter `i` ranges from `0` (the index of the string's first character) to string length - 1 (index of the last character). When the counter value equals the string length, the expression becomes false and the loop ends.

So, the previous example may also be written with a `for` for an identical result.

```
const name = "Sarah";
for (let i = 0; i < name.length; i++) {
    console.log(name[i]);
}
```

Recent JavaScript evolution has introduced yet another option to iterate over a string: the `for-of` loop. The previous example may also be written:.

```
const name = "Sarah";
for (const letter of name) {
    console.log(letter);
}
```

If the index is not needed inside the loop, this syntax is arguably the simplest one.

Searching inside a string

Looking for particular values inside a string is a common task.

The `indexOf()` takes as a parameter the searched value. If that value is found inside the string, it returns the index of the first occurrence of the value. Otherwise, it returns -1.

```
const song = "Honky Tonk Women";
console.log(song.indexOf("onk")); // 1
console.log(song.indexOf("Onk")); // -1 because of case mismatch
```

When searching for a value at the beginning or end of a string, you may also use the `startsWith()` and `endsWith()` methods. Both return either `true` or `false`, depending on whether the value is found or not.

```
const song = "Honky Tonk Women";

console.log(song.startsWith("Honk")); // true
console.log(song.startsWith("Tonk")); // false because of case sensitivity

console.log(song.endsWith("men")); // true
console.log(song.endsWith("Men")); // false because of case sensitivity
```

Breaking a string into parts

Sometimes a string is made of several parts separated by a particular value. In that case, it's easy to obtain the individual parts by using the `split()` method. It takes as a parameter the separator and returns an array containing the parts.

```
const monthList = "Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov,Dec";
const months = monthList.split(",");
console.log(months[0]);  // "Jan"
console.log(months[11]); // "Dec"
```

More on Strings

- Mozilla Developer Network JavaScript Reference: String[3]
- W3 Schools Strings[4]
- W3 Schools String Methods[5]
- W3 Schools String Reference[6]
- Quirksmode Strings[7]

Coding time!

Word info

Write a program that asks you for a word then shows its length, lowercase and uppercase values.

Vowel count

Improve the previous program so that it also shows the number of vowels inside the word.

[3]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String
[4]https://www.w3schools.com/js/js_strings.asp
[5]https://www.w3schools.com/js/js_string_methods.asp
[6]https://www.w3schools.com/jsref/jsref_obj_string.asp
[7]http://www.quirksmode.org/js/strings.html

Backwards word

Improve the previous program so that it shows the word written backwards.

Palindrome

Improve the previous program to check if the word is a palindrome. A palindrome is a word or sentence that’s spelled the same way both forward and backward, ignoring punctuation, case, and spacing. Punctuation and spacing may not be taken into account here.

> `"radar"` should be detected as a palindrome, `"Radar"` too.

9. Understand object-oriented programming

A few chapters ago, you learned how to create your first objects in JavaScript. Now it's time to better understand how to work with them.

TL;DR

- **Object-Oriented Programming**, or OOP, is a programming paradigm[1] that uses objects containing both **data** and **behavior** to create programs.
- A **class** is an object-oriented abstraction for an idea or a concept manipulated by a program. It offers a convenient syntax to create objects representing this concept.
- A JavaScript class is defined with the `class` keyword. It can only contain **methods**. The `constructor()` method, called during object creation, is used to initialize the object, often by giving it some data properties. Inside methods, the `this` keyword represents **the object on which the method was called**.

```
class MyClass {
    constructor(param1, param2, ...) {
        this.property1 = param1;
        this.property2 = param2;
        // ...
    }
    method1(/* ... */) {
        // ...
    }
    method2(/* ... */) {
        // ...
    }
    // ...
}
```

- Objects are created from a class with the `new` operator. It calls the class constructor to initialize the newly created object.

[1] https://en.wikipedia.org/wiki/Programming_paradigm

```
const myObject = new MyClass(arg1, arg2, ...);
// ...
```

- JavaScript's OOP model is based on **prototypes**. Any JavaScript object has an internal property which is a link (a **reference**) to another object: its prototype. Prototypes are used to share properties and delegate behavior between objects.
- When trying to access a property that does not exist in an object, JavaScript tries to find this property in the **prototype chain** of this object: its prototype, then its prototype's own prototype, and so on.
- There are several ways to create and link JavaScript objects through prototypes. One is to use the `Object.create()` method.

```
// Create an object linked to myPrototypeObject
const myObject = Object.create(myPrototypeObject);
```

- The JavaScript `class` syntax is another, arguably more convenient way to create relationships between objects. It emulates the class-based OOP model found in languages like C++, Java or C#. It is, however, just **syntactic sugar** on top of JavaScript's own prototype-based OOP model.

Context: a multiplayer RPG

As a reminder, here's the code for our minimalist RPG taken from a previous chapter. it creates an object literal named `aurora` with three properties (`name`, `health` and `strength`) and a `describe()` method.

```
const aurora = {
  name: "Aurora",
  health: 150,
  strength: 25,
  xp: 0,

  // Return the character description
  describe() {
    return `${this.name} has ${this.health} health points, ${this.strength} a\
s strength and ${this.xp} XP points`;
  }
};

// Aurora is harmed by an arrow
aurora.health = aurora.health - 20;

// Aurora equips a strength necklace
```

```
aurora.strength = aurora.strength + 10;

// Aurora learn a new skill
aurora.xp = aurora.xp + 15;

console.log(aurora.describe());
```

To make the game more interesting, we'd like to have more characters in it. So here comes Glacius, Aurora's fellow.

```
const glacius = {
  name: "Glacius",
  health: 139,
  strength: 30,
  xp: 0,

  // Return the character description
  describe() {
    return `${this.name} has ${this.health} health points, ${this.strength} a\
s strength and ${this.xp} XP points`;
  }
};
```

Our two characters are strikingly similar. They share the same properties, with the only difference being some property values.

You should already be aware that code duplication is dangerous and should generally be avoided. We must find a way to share what's common to our characters.

JavaScript classes

Most object-oriented languages use classes as **abstractions** for the ideas or concepts manipulated by a program. A **class** is used to create objects representing a concept. It offers a convenient syntax to give both **data** and **behavior** to these objects.

JavaScript is no exception and supports programming with classes (but with a twist – more on that later).

Creating a class

Our example RPG deals with characters, so let's create a `Character` class to express what a character is.

```
class Character {
  constructor(name, health, strength) {
    this.name = name;
    this.health = health;
    this.strength = strength;
    this.xp = 0; // XP is always zero for new characters
  }
  // Return the character description
  describe() {
    return `${this.name} has ${this.health} health points, ${this.strength} a\
s strength and ${this.xp} XP points`;
  }
}
```

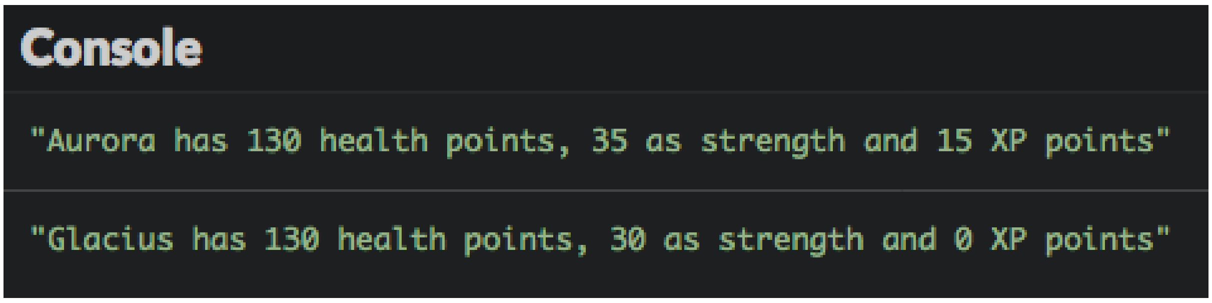
This example demonstrates several key facts about JavaScript classes:

- A class is created with the `class` keyword, followed by the name of the class (usually starting with a uppercase letter).
- Contrary to object literals, there is no separating punctuation between the elements inside a class.
- A class can only contains **methods**, not data properties.
- Just like with object literals, the `this` keyword is automatically set by JavaScript inside a method and represents **the object on which the method was called**.
- A special method named `constructor()` can be added to a class definition. It is called during object creation and is often used to give it data properties.

Using a class

Once a class is defined, you can use it to create objects. Check out the rest of the program.

```
const aurora = new Character("Aurora", 150, 25);
const glacius = new Character("Glacius", 130, 30);

// Aurora is harmed by an arrow
aurora.health = aurora.health - 20;

// Aurora equips a strength necklace
aurora.strength = aurora.strength + 10;

// Aurora learn a new skill
aurora.xp = aurora.xp + 15;

console.log(aurora.describe());
console.log(glacius.describe());
```



```
Console
"Aurora has 130 health points, 35 as strength and 15 XP points"
"Glacius has 130 health points, 30 as strength and 0 XP points"
```

Execution result

The `aurora` and `glacius` objects are created as characters with the `new` operator. This statement calls the class constructor to initialize the newly created object. After creation, an object has access to the properties defined inside the class.

Here's the canonical syntax for creating an object using a class.

```
class MyClass {
    constructor(param1, param2, ...) {
        this.property1 = param1;
        this.property2 = param2;
        // ...
    }
    method1(/* ... */) {
        // ...
    }
    method2(/* ... */) {
        // ...
    }
    // ...
}

const myObject = new MyClass(arg1, arg2, ...);
myObject.method1(/* ... */);
// ...
```

Under the hood: objects and prototypes

If you come from another programming background, chances are you already encountered classes and feel familiar with them. But as you'll soon discover, JavaScript classes are not quite like their C++, Java or C# counterparts.

JavaScript's object-oriented model

To create relationships between objects, JavaScript uses **prototypes**.

In addition to its own particular properties, any JavaScript object has an internal property which is a link (known as a **reference**) to another object called its **prototype**. When trying to access a property that does not exist in an object, JavaScript tries to find this property in the prototype of this object.

Here's an example (borrowed from Kyle Simpson's great book series You Don't Know JS[2]).

```
const anObject = {
    myProp: 2
};

// Create anotherObject using anObject as a prototype
const anotherObject = Object.create(anObject);

console.log(anotherObject.myProp); // 2
```

In this example, the JavaScript statement `Object.create()` is used to create the object `anotherObject` with object `anObject` as its prototype.

```
// Create an object linked to myPrototypeObject
const myObject = Object.create(myPrototypeObject);
```

When the statement `anotherObject.myProp` is run, the `myProp` property of `anObject` is used since `myProp` doesn't exist in `anotherObject`.

If the prototype of an object does not have a desired property, then research continues in its own prototype until we get to the end of the **prototype chain**. If the end of this chain is reached without having found the property, access to it returns the value `undefined`.

```
const anObject = {
    myProp: 2
};

// Create anotherObject using anObject as a prototype
const anotherObject = Object.create(anObject);

// Create yetAnotherObject using anotherObject as a prototype
const yetAnotherObject = Object.create(anotherObject);

// myProp is found in yetAnotherObject's prototype chain (in anObject)
console.log(yetAnotherObject.myProp); // 2

// myOtherProp can't be found in yetAnotherObject's prototype chain
console.log(yetAnotherObject.myOtherProp); // undefined
```

This type of relationship between JavaScript objects is called **delegation**: an object delegates part of its operation to its prototype.

[2]https://github.com/getify/You-Dont-Know-JS/blob/master/this%20%26%20object%20prototypes/ch5.md

The true nature of JavaScript classes

In *class-based* object-oriented languages like C++, Java and C#, classes are static **blueprints** (templates). When a object is created, the methods and properties of the class are copied into a new entity, called an **instance**. After instantiation, the newly created object has no relation whatsoever with its class.

JavaScript's object-oriented model is based on prototypes, *not* classes, to share properties and delegate behavior between objects. In JavaScript, a class is itself an object, not a static blueprint. "Instanciating" a class creates a new object linked to a prototype object. Regarding classes behavior, the JavaScript language is quite different from C++, Java or C#, but close to other object-oriented languages like Python, Ruby and Smalltalk.

The JavaScript `class` syntax is merely a more convenient way to create relationships between objects through prototypes. Classes were introduced to emulate the class-based OOP model on top of JavaScript's own prototype-based model. It's an example of what programmers call syntactic sugar[3].

> The usefulness of the `class` syntax is a pretty heated debate in the JavaScript community.

Object-oriented programming

Now back to our RPG, which is still pretty boring. What does it lack? Monsters and fights, of course!

Here's how a fight will be handled: if attacked, a character sees their life points decrease from the strength of the attacker. If its health value fall below zero, the character is considered dead and cannot attack anymore. Its vanquisher receives a fixed number of 10 experience points.

First, let's add the possibility for our characters to fight one another. Since it's a shared ability, we define it as a method named `attack()` in the `Character` class.

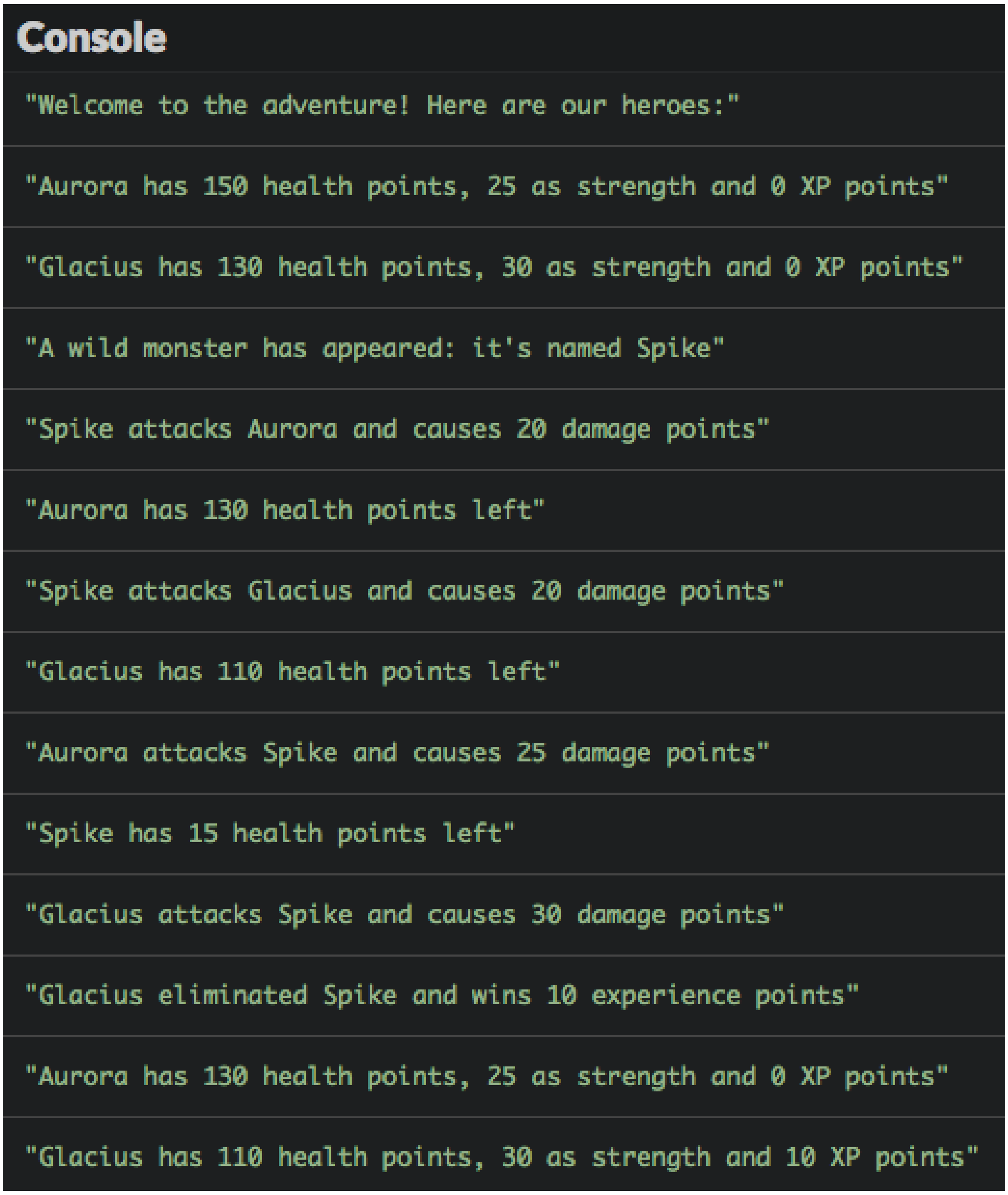
```
class Character {
  constructor(name, health, strength) {
    this.name = name;
    this.health = health;
    this.strength = strength;
    this.xp = 0; // XP is always zero for new characters
  }
  // Attack a target
  attack(target) {
    if (this.health > 0) {
      const damage = this.strength;
      console.log(`${this.name} attacks ${target.name} and causes ${damage} d\
amage points`);
```

[3] https://en.wikipedia.org/wiki/Syntactic_sugar

```
      target.health -= damage;
      if (target.health > 0) {
        console.log(`${target.name} has ${target.health} health points left`);
      } else {
        target.health = 0;
        const bonusXP = 10;
        console.log(`${this.name} eliminated ${target.name} and wins ${bonusX\
P} experience points`);
        this.xp += bonusXP;
      }
    } else {
      console.log(`${this.name} can't attack (they've been eliminated)`);
    }
  }
  // Return the character description
  describe() {
    return `${this.name} has ${this.health} health points, ${this.strength} a\
s strength and ${this.xp} XP points`;
  }
}
```

Now we can introduce a monster in the game and make it fight our players. Here's the rest of the final code of our RPG.

```
const aurora = new Character("Aurora", 150, 25);
const glacius = new Character("Glacius", 130, 30);

console.log("Welcome to the adventure! Here are our heroes:");
console.log(aurora.describe());
console.log(glacius.describe());

const monster = new Character("Spike", 40, 20);
console.log("A wild monster has appeared: it's named " + monster.name);

monster.attack(aurora);
monster.attack(glacius);
aurora.attack(monster);
glacius.attack(monster);

console.log(aurora.describe());
console.log(glacius.describe());
```



```
Console
"Welcome to the adventure! Here are our heroes:"
"Aurora has 150 health points, 25 as strength and 0 XP points"
"Glacius has 130 health points, 30 as strength and 0 XP points"
"A wild monster has appeared: it's named Spike"
"Spike attacks Aurora and causes 20 damage points"
"Aurora has 130 health points left"
"Spike attacks Glacius and causes 20 damage points"
"Glacius has 110 health points left"
"Aurora attacks Spike and causes 25 damage points"
"Spike has 15 health points left"
"Glacius attacks Spike and causes 30 damage points"
"Glacius eliminated Spike and wins 10 experience points"
"Aurora has 130 health points, 25 as strength and 0 XP points"
"Glacius has 110 health points, 30 as strength and 10 XP points"
```

Execution result

The previous program is a short example of **Object-Oriented Programming** (in short: OOP), a programming paradigm[4] (a programming style) based on objects containing both data and behavior.

[4]https://en.wikipedia.org/wiki/Programming_paradigm

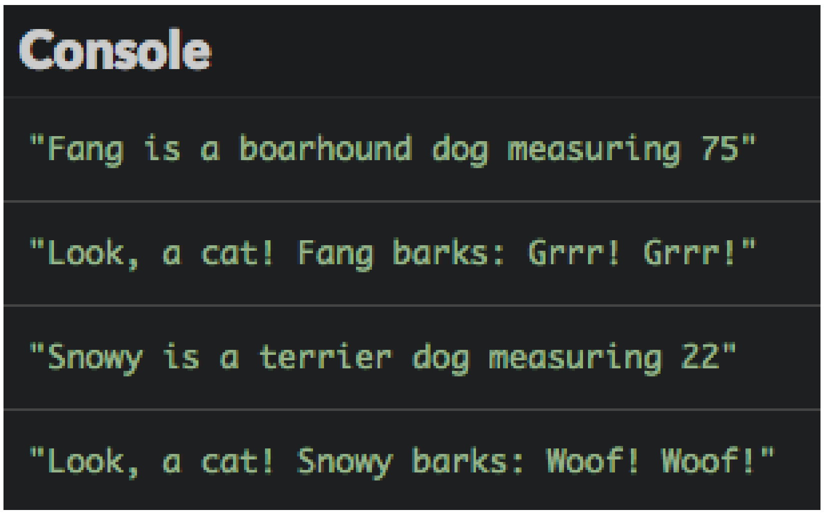
Coding time!

Dogs

Complete the following program to add the definition of the `Dog` class.

> Dogs taller than 60 make `"Grrr! Grrr!"` when they bark, other ones make `"Woof! Woof!"`.

```
// TODO: define the Dog class here

const fang = new Dog("Fang", "boarhound", 75);
console.log(`${fang.name} is a ${fang.species} dog measuring ${fang.size}`);
console.log(`Look, a cat! ${fang.name} barks: ${fang.bark()}`);

const snowy = new Dog("Snowy", "terrier", 22);
console.log(`${snowy.name} is a ${snowy.species} dog measuring ${snowy.size}`\
);
console.log(`Look, a cat! ${snowy.name} barks: ${snowy.bark()}`);
```



```
Console
"Fang is a boarhound dog measuring 75"
"Look, a cat! Fang barks: Grrr! Grrr!"
"Snowy is a terrier dog measuring 22"
"Look, a cat! Snowy barks: Woof! Woof!"
```

Execution result

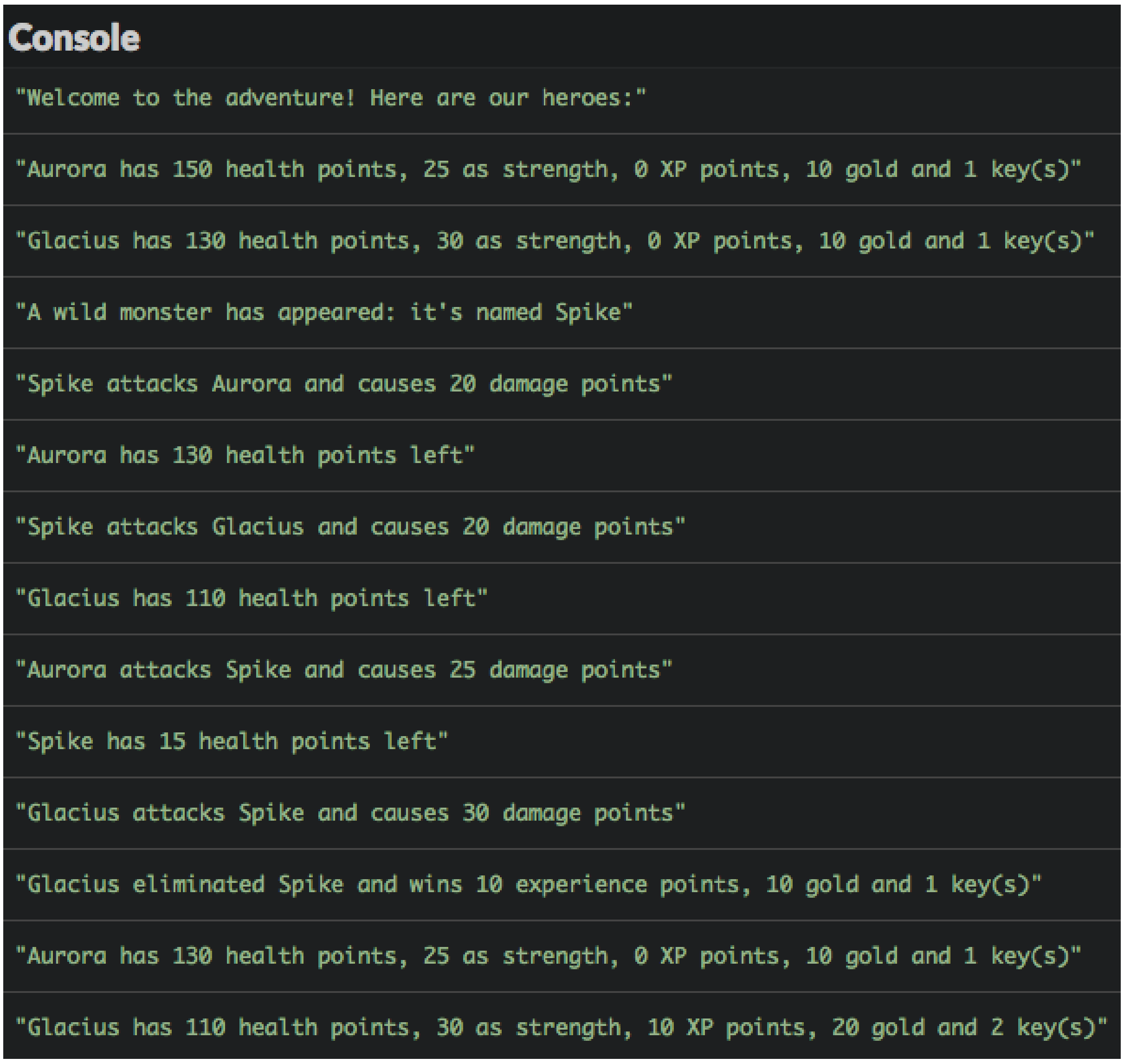
Character inventory

Improve the example RPG to add character inventory management according to the following rules:

- A character's inventory contains a number of gold and a number of keys.
- Each character begins with 10 gold and 1 key.

- The character description must show the inventory state.
- When a character slays annother one, the victim's inventory goes to its vanquisher.

Here's the expected execution result.



```
Console
"Welcome to the adventure! Here are our heroes:"
"Aurora has 150 health points, 25 as strength, 0 XP points, 10 gold and 1 key(s)"
"Glacius has 130 health points, 30 as strength, 0 XP points, 10 gold and 1 key(s)"
"A wild monster has appeared: it's named Spike"
"Spike attacks Aurora and causes 20 damage points"
"Aurora has 130 health points left"
"Spike attacks Glacius and causes 20 damage points"
"Glacius has 110 health points left"
"Aurora attacks Spike and causes 25 damage points"
"Spike has 15 health points left"
"Glacius attacks Spike and causes 30 damage points"
"Glacius eliminated Spike and wins 10 experience points, 10 gold and 1 key(s)"
"Aurora has 130 health points, 25 as strength, 0 XP points, 10 gold and 1 key(s)"
"Glacius has 110 health points, 30 as strength, 10 XP points, 20 gold and 2 key(s)"
```

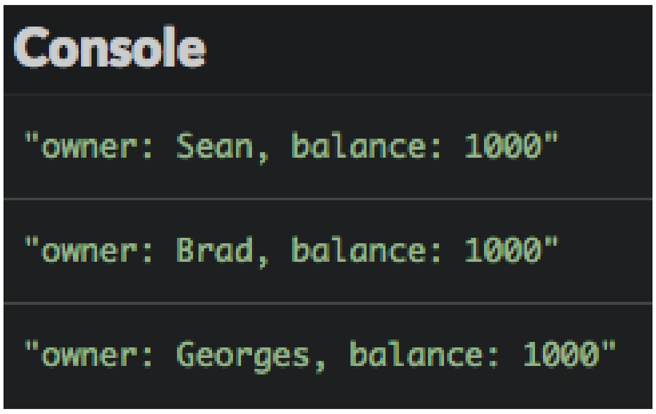
Execution result

Account list

Let's build upon a previous account object exercise. A bank account is still defined by:

- A `name` property.
- A `balance` property, initially set to `0`.
- A `credit` method adding the value passed as an argument to the account balance.
- A `describe` method returning the account description.

Write a program that creates three accounts: one belonging to Sean, another to Brad and the third one to Georges. These accounts are stored in an array. Next, the program credits 1000 to each account and shows its description.


Console

```
"owner: Sean, balance: 1000"
"owner: Brad, balance: 1000"
"owner: Georges, balance: 1000"
```

Execution result

10. Discover functional programming

Object-oriented programming, albeit quite popular, is not the only way to create programs. This chapter will introduce you to another important paradigm: functional programming.

TL;DR

- **Functional programming** is about writing programs by combining functions expressing *what* the program should do, rather than *how* to do it (which is the imperative way).
- The **state** of a program is the value of its **global variables** at a given time. A goal of functional programming is to minimize state **mutations** (changes) that make the code harder to understand. Some possible solutions are declaring variables with `const` instead of `let`, splitting the code into functions and favor local variables over global ones.
- A **pure function** depends solely in its inputs for computing its outputs and has no **side effect**. Pure functions are easier to understand, combine together, and debug. Functional programming favors the use of pure functions whenever possible.
- The `map()`, `filter()` and `reduce()` methods can replace loops for array traversal and let you program with arrays in a functional way.
- JavaScript functions can be passed around just like any other value: they are **first-class citizens**, enabling functional programming. A function that operates on another function (taking it as an parameter or returning it) is called an **higher-order function**.
- JavaScript is a **multi-paradigm** language: you can write programs using an imperative, object-oriented or functional programming style.

Context: a movie list

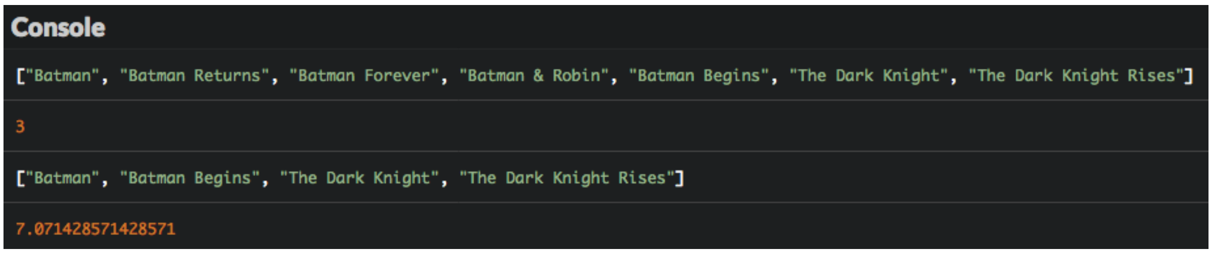
In this chapter, we'll start with an example program and improve it little by little, without adding any new functionality. This important programmer task is called **refactoring**.

Our initial program is about recent Batman movies. The data comes under the form of an array of objects, each object describing a movie.

```
const movieList = [{
  title: "Batman",
  year: 1989,
  director: "Tim Burton",
  imdbRating: 7.6
}, {
  title: "Batman Returns",
  year: 1992,
  director: "Tim Burton",
  imdbRating: 7.0
}, {
  title: "Batman Forever",
  year: 1995,
  director: "Joel Schumacher",
  imdbRating: 5.4
}, {
  title: "Batman & Robin",
  year: 1997,
  director: "Joel Schumacher",
  imdbRating: 3.7
}, {
  title: "Batman Begins",
  year: 2005,
  director: "Christopher Nolan",
  imdbRating: 8.3
}, {
  title: "The Dark Knight",
  year: 2008,
  director: "Christopher Nolan",
  imdbRating: 9.0
}, {
  title: "The Dark Knight Rises",
  year: 2012,
  director: "Christopher Nolan",
  imdbRating: 8.5
}];
```

And here is the rest of the program that uses this data to show some results about the movies. Check it out, it should be pretty self-explanatory.

```
// Get movie titles
const titles = [];
for (movie of movieList) {
  titles.push(movie.title);
}
console.log(titles);

// Count movies by Christopher Nolan
const nolanMovieList = [];
for (movie of movieList) {
  if (movie.director === "Christopher Nolan") {
    nolanMovieList.push(movie);
  }
}
console.log(nolanMovieList.length);

// Get titles of movies with an IMDB rating greater or equal to 7.5
const bestTitles = [];
for (movie of movieList) {
  if (movie.imdbRating >= 7.5) {
    bestTitles.push(movie.title);
  }
}
console.log(bestTitles);

// Compute average movie rating of Christopher Nolan's movies
let ratingSum = 0;
let averageRating = 0;
for (movie of nolanMovieList) {
  ratingSum += movie.imdbRating;
}
averageRating = ratingSum / movieList.length;
console.log(averageRating);
```



```
["Batman", "Batman Returns", "Batman Forever", "Batman & Robin", "Batman Begins", "The Dark Knight", "The Dark Knight Rises"]
3
["Batman", "Batman Begins", "The Dark Knight", "The Dark Knight Rises"]
7.071428571428571
```

Execution result

Program state

The previous program is an example of what is called **imperative programming**. In this paradigm, the programmer gives orders to the computer through a series of statements that modify the program state. Imperative programming focuses on describing *how* a program operates.

The concept of state is an important one. The **state** of a program is the value of its **global variables** (variables accessible everythere in the code) at a given time. In our example, the values of `movieList`, `titles`, `nolanMovieCount`, `bestTitles`, `ratingSum` and `averageRating` form the state of the program. Any assignment to one of these variables is a state change, often called a **mutation**.

In imperative programming, the state can be modified anywhere in the source code. This is convenient, but can also lead to nasty bugs and maintenance headaches. As the program grows in size and complexity, it's becoming easier for the programmer to mutate a part of the state by mistake, and harder to monitor state modifications.

Limiting mutations with `const` variables

In order to decrease the risk of accidental state mutation, a first step is to favor `const` over `let` whenever applicable for variable declarations. A variable declared with the `const` keyword cannot be further reassigned. Array and object content can still be mutated, though. Check the following code for details.

```
const n = 10;
const fruit = "Banana";
const obj = {
  myProp: 2
};
const animals = ["Elephant", "Turtle"];

obj.myProp = 3;          // Mutating a property is OK even for a const object
obj.myOtherProp = "abc"; // Adding a new property is OK even for a const obje\
ct
animals.push("Gorilla"); // Updating content is OK even for a const array

n++;                  // Illegal
fruit = "orange";     // Illegal
obj = {};             // Illegal
animals = ["Bee"]; // Illegal
```

Splitting the program into functions

Another solution is to split the source code into subroutines called procedures or **functions**. This approach is called **procedural programming** and has the benefit of transforming some variables into **local variables**, which are only visible in the subroutine code.

Let’s try to introduce some functions in our code.

```
// Get movie titles
function titles() {
  const titles = [];
  for (movie of movieList) {
    titles.push(movie.title);
  }
  return titles;
}

// Get movies by Christopher Nolan
function nolanMovies() {
  for (movie of movieList) {
    if (movie.director === "Christopher Nolan") {
      nolanMovieList.push(movie);
    }
  }
}

// Get titles of movies with an IMDB rating greater or equal to 7.5
function bestTitles() {
  const bestTitles = [];
  for (movie of movieList) {
    if (movie.imdbRating >= 7.5) {
      bestTitles.push(movie.title);
    }
  }
  return bestTitles;
}

// Compute average rating of Christopher Nolan's movies
function averageNolanRating() {
  let ratingSum = 0;
  for (movie of nolanMovieList) {
    ratingSum += movie.imdbRating;
  }
  return ratingSum / nolanMovieList.length;
}

const nolanMovieList = [];

console.log(titles());
nolanMovies();
console.log(nolanMovieList.length);
console.log(bestTitles());
console.log(averageNolanRating());
```

The state of our program is now limited to two variables: `movieList` and `nolanMovieList` (the latter being necessary in functions `nolanMovies()` and `averageNolanRating()`). The other variables are now local to the functions they are used into, which limit the possibility of an accidental state mutation.

Also, this version of the program is easier to understand than the previous one. Functions with appropriate names help describe a program's behavior. Comments are now less necessary than before.

Pure functions

Merely introducing some functions in a program is not enough to follow the functional programming paradigm. Whenever possible, we also need to use pure functions.

A **pure function** is a function has the following characteristics:

- Its outputs depend solely on its inputs.
- It has no side effect.

A **side effect** is a change in program state or an interaction with the outside world. A database access or a `console.log()` statement are examples of side effects.

Given the same data, a pure function will always produce the same result. By design, a pure function is independent from the program state and must not access it. Such a function must accept **parameters** in order to do something useful. The only way for a function without parameters to be pure is to return a constant value.

Pure functions are easier to understand, combine together, and debug: contrary to their *impure* counterparts, there's no need to look outside the function body to reason about it. Still, a number of side effects are necessary in any program, like showing output to the user or updating a database. In functional programming, the name of the game is to create those side effects only in some dedicated and clearly identified parts of the program. The rest of the code should be written as pure functions.

Let's refactor our example code to introduce pure functions.

```
// Get movie titles
function titles(movies) {
  const titles = [];
  for (movie of movies) {
    titles.push(movie.title);
  }
  return titles;
}
```

```
// Get movies by Christopher Nolan
function nolanMovies(movies) {
  const nolanMovies = [];
  for (movie of movies) {
    if (movie.director === "Christopher Nolan") {
      nolanMovies.push(movie);
    }
  }
  return nolanMovies;
}

// Get titles of movies with an IMDB rating greater or equal to 7.5
function bestTitles(movies) {
  const bestTitles = [];
  for (movie of movies) {
    if (movie.imdbRating >= 7.5) {
      bestTitles.push(movie.title);
    }
  }
  return bestTitles;
}

// Compute average rating of a movie list
function averageRating(movies) {
  let ratingSum = 0;
  for (movie of movies) {
    ratingSum += movie.imdbRating;
  }
  return ratingSum / movies.length;
}

console.log(titles(movieList));
const nolanMovieList = nolanMovies(movieList);
console.log(nolanMovieList.length);
console.log(bestTitles(movieList));
console.log(averageRating(nolanMovieList));
```

Since we only do refactoring, the program output is still the same.

The program state (`movieList` and `nolanMovieList`) hasn't changed. However, all our functions are now pure: instead of accessing the state, they use parameters to achieve their desired behavior. As an added benefit, the function `averageRating()` can now compute the average rating of any movie list: it has become more **generic**.

Array operations

Functional programming is about writing programs by combining functions expressing *what* the program should do, rather than *how* to do it. JavaScript offers several array-related methods that favor a functional programming style.

The `map()` method

The `map()` method takes an array as a parameter and creates a new array with the results of calling a provided function on every element in this array. A typical use of `map()` is to replace a loop for array traversal.

Let's see `map()` in action.

```
const numbers = [1, 5, 10, 15];
// The associated function multiply each array number by 2
const doubles = numbers.map(x => x * 2);

console.log(numbers); // [1, 5, 10, 15] (no change)
console.log(doubles); // [2, 10, 20, 30]
```

Here's how our `titles()` could be rewritten using `map()`. Look how the function code is now more concise and expressive.

```
// Get movie titles
function titles(movies) {
  /* Previous code
  const titles = [];
  for (movie of movies) {
    titles.push(movie.title);
  }
  return titles;
  */

  // Return a new array containing only movie titles
  return movies.map(movie => movie.title);
}
```

The `filter()` method

The `filter()` method offers a way to test every element of an array against a provided function. Only elements that pass this test are added in the returned array.

Here's an example of using `filter()`.

```
const numbers = [1, 5, 10, 15];
// Keep only the number greater or equal to 10
const bigOnes = numbers.filter(x => x >= 10);

console.log(numbers); // [1, 5, 10, 15] (no change)
console.log(bigOnes); // [10, 15]
```

We can use this method in the `nolanMovies()` function.

```
// Get movies by Christopher Nolan
function nolanMovies(movies) {
  /* Previous code
  const nolanMovies = [];
  for (movie of movies) {
    if (movie.director === "Christopher Nolan") {
      nolanMovies.push(movie);
    }
  }
  return nolanMovies;
  */

  // Return a new array containing only movies by Christopher Nolan
  return movies.filter(movie => movie.director === "Christopher Nolan");
}
```

The `map()` and `filter()` method can be used together to achieve powerful effects. Look at this new version of the `bestTitles()` function.

```
// Get titles of movies with an IMDB rating greater or equal to 7.5
function bestTitles(movies) {
  /* Previous code
  onst bestTitles = [];
  for (movie of movies) {
    if (movie.imdbRating >= 7.5) {
      bestTitles.push(movie.title);
    }
  }
  return bestTitles;
  */

  // Filter movies by IMDB rating, then creates a movie titles array
  return movies.filter(movie => movie.imdbRating >= 7.5).map(movie => movie.t\
itle)
}
```

The `reduce()` method

The `reduce()` method applies a provided function to each array element in order to *reduce* it to one value. This method is typically used to perform calculations on an array.

Here's an example of reducing an array to the sum of its values.

```
const numbers = [1, 5, 10, 15];
// Compute the sum of array elements
const sum = numbers.reduce((acc, value) => acc + value, 0);

console.log(numbers); // [1, 5, 10, 15] (no change)
console.log(sum);     // 31
```

The `reduce()` method can take several parameters:

- The first one is the function associated to `reduce()` and called for each array element takes two parameters: the first is an **accumulator** which contains the accumulated value previously returned by the last invocation of the function. The other function parameter is the array element.
- The second one is the initial value of the accumulator (often `0`).

Here's how to apply `reduce()` to caculate the average rating of a movie list.

```
// Compute average rating of a movie list
function averageRating(movies) {
  /* Previous code
  let ratingSum = 0;
  for (movie of movies) {
    ratingSum += movie.imdbRating;
  }
  return ratingSum / movies.length;
  */

  // Compute the sum of all movie IMDB ratings
  const ratingSum = movies.reduce((acc, movie) => acc + movie.imdbRating, 0);
  return ratingSum / movies.length;
}
```

Another possible solution is to compute the rating sum by using `map()` before reducing an array containing only movie ratings.

```
// ...
// Compute the sum of all movie IMDB ratings
const ratingSum = movies.map(movie => movie.imdbRating).reduce((acc, value) =\
> acc + value, 0);
// ...
```

Higher-order functions

Throughout this chapter, we have leveraged the fact that JavaScript functions can be passed around just like any other value. We say that functions are **first-class citizens** in JavaScript, which means that they are treated equal to other types.

Thanks to their first-class citizenry, functions can be combined together, rendering programs even more expressive and enabling a truly functional programming style. A function that takes another function as a parameter or returns another function is called an **higher-order function**.

Check out this final version of our example program.

```
const titles = movies => movies.map(movie => movie.title);
const byNolan = movie => movie.director === "Christopher Nolan";
const filter = (movies, fct) => movies.filter(fct);
const goodRating = movie => movie.imdbRating >= 7.5;
const ratings = movies => movies.map(movie => movie.imdbRating);
const average = array => array.reduce((sum, value) => sum + value, 0) / array\
.length;

console.log(titles(movieList));
const nolanMovieList = filter(movieList, byNolan);
console.log(nolanMovieList.length);
console.log(titles(filter(movieList, goodRating)));
console.log(average(ratings(nolanMovieList)));
```

We have defined helper functions that we combine to achieve the desired behaviour. The code is concise and self-describing. Since it takes the filtering function as a parameter, our own `filter()` function is an example of an higher-order function.

JavaScript: a multi-paradigm language

The JavaScript language is full of paradoxes. It has famously been invented in ten days[1], and is now enjoying a popularity almost unique in programming history. Its syntax borrows heavily from maintream imperative languages like C or Java, but its design principles are closer to functional languages like Scheme[2].

JavaScript's multi-paradigm nature means you can write imperative, object-oriented or functional code, choosing the right tool for the job and leveraging your previous programming experience. As always, diversity is a source of flexibility and ultimately a strength.

[1]https://www.w3.org/community/webed/wiki/A_Short_History_of_JavaScript
[2]https://en.wikipedia.org/wiki/Scheme_(programming_language)

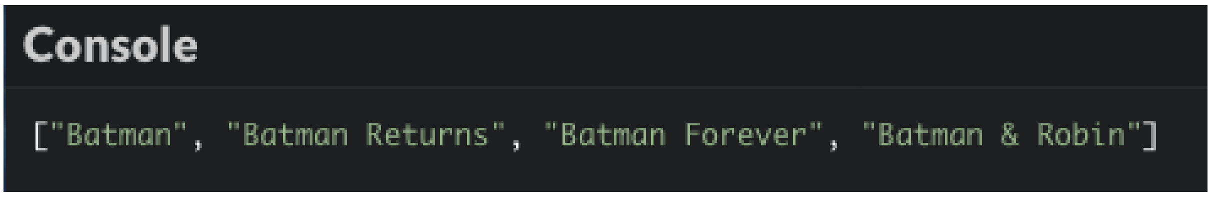
Coding time!

Older movies

Improve the example movie program from above so that it shows the titles of movies released before year 2000, using functional programming.

```
const movieList = [{
  title: "Batman",
  year: 1989,
  director: "Tim Burton",
  imdbRating: 7.6
}, {
  title: "Batman Returns",
  year: 1992,
  director: "Tim Burton",
  imdbRating: 7.0
}, {
  title: "Batman Forever",
  year: 1995,
  director: "Joel Schumacher",
  imdbRating: 5.4
}, {
  title: "Batman & Robin",
  year: 1997,
  director: "Joel Schumacher",
  imdbRating: 3.7
}, {
  title: "Batman Begins",
  year: 2005,
  director: "Christopher Nolan",
  imdbRating: 8.3
}, {
  title: "The Dark Knight",
  year: 2008,
  director: "Christopher Nolan",
  imdbRating: 9.0
}, {
  title: "The Dark Knight Rises",
  year: 2012,
  director: "Christopher Nolan",
  imdbRating: 8.5
}];

// TODO: Make an array of the titles of movies released before 2000
```

```
console.log ( moviesBefore2000 );
```



```
Console
["Batman", "Batman Returns", "Batman Forever", "Batman & Robin"]
```

Execution result

Government forms

Complete the following program to compute and show the names of political forms ending with "cy".

```
const governmentForms = [{
  name: "Plutocracy",
  definition: "Rule by the weathly"
},{
  name: "Oligarchy",
  definition: "Rule by a small number of people"
},{
  name: "Kleptocracy",
  definition: "Rule by the thieves"
},{
  name: "Theocracy",
  definition: "Rule by a religious elite"
}, {
  name: "Democracy",
  definition: "Rule by the people"
}, {
  name: "Autocracy",
  definition: "Rule by a single person"
}];

// TODO: compute the formsEndingWithCy array

// Should show ["Plutocracy", "Kleptocracy", "Theocracy", "Democracy", "Autoc\
racy"]
console.log(formsEndingWithCy);
```

Arrays sum

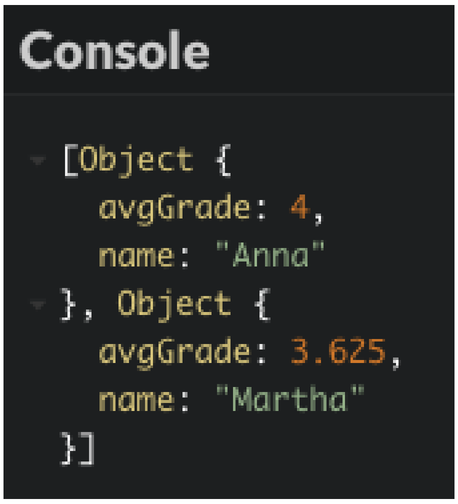
Complete the following program to compute and show the total sum of the values in each of the arrays.

```
const arrays = [
  [1, 4],
  [11],
  [3, 5, 7]
];

// TODO: compute the value of the arraysSum variable

console.log(arraysSum); // Should show 31
```

Student results

Here's a program that shows female students results (name and average grade). Refactor it using functional programming. Execution result must stay the same.

```
const students = [{
  name: "Anna",
  sex: "f",
  grades: [4.5, 3.5, 4]
}, {
  name: "Dennis",
  sex: "m",
  country: "Germany",
  grades: [5, 1.5, 4]
}, {
  name: "Martha",
  sex: "f",
  grades: [5, 4, 2.5, 3]
}, {
  name: "Brock",
  sex: "m",
  grades: [4, 3, 2]
}];

// Compute female student results
const femaleStudentsResults = [];
for (student of students) {
  if (student.sex === "f") {
    let gradesSum = 0;
    for (grade of student.grades) {
      gradesSum += grade;
    }
    let averageGrade = gradesSum / student.grades.length;
    femaleStudentsResults.push({
      name: student.name,
      avgGrade: averageGrade
    });
  }
}

console.log(femaleStudentsResults);
```



```
Console

[Object {
  avgGrade: 4,
  name: "Anna"
}, Object {
  avgGrade: 3.625,
  name: "Martha"
}]
```

Execution result

11. Project: a social news program

Now that you've discovered the basics of programming, let's go ahead and build a real project.

Objective

The goal of this project is to build a basic social news program. Its user will be able to show a list of links and add new ones.

Functional requirements

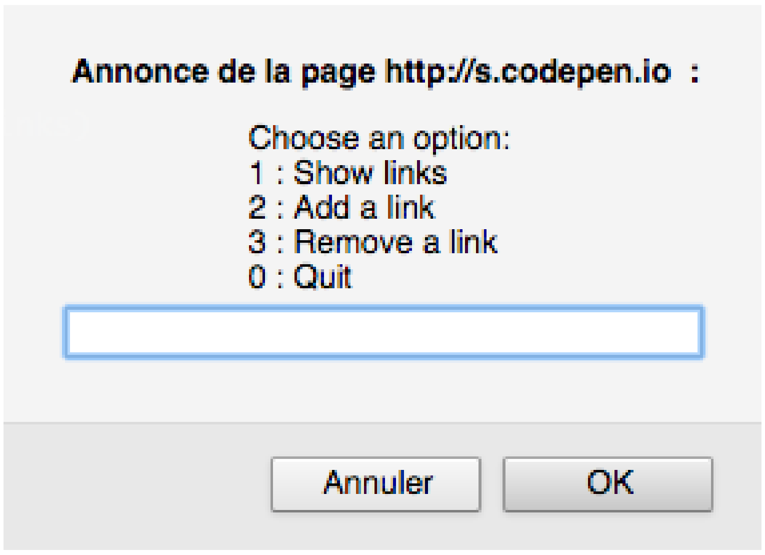
- A link is defined by its title, its URL and its author (submitter).
- At launch, the program displays a start menu with the possible actions in an alert window and asks the user for his choice. Possible actions are:
  - Show the list of links.
  - Add a new link.
  - Remove an existing link.
  - Quit the program.
- Showing the list of links displays the index (rank) and the properties of each link in an alert window, or a message in the absence of any link.
- When adding a link, the program asks the user for the new link properties (title, URL and author). The link is then created. Subsequently, it must appear in the showed links.
- If the new link URL does not start with `"http://"` or `"https://"`, the program adds `"http://"` at its beginning.
- When removing a link, the user is asked for the link index until it is correct. The associated link is then removed. Subsequently, it must disappear from the showed links. Removing a link is not possible if there are no existing links.
- After an action is performed, the start menu is shown again. This goes on until the user chooses to quit the program.

Technical requirements

- All your code should be correctly indented.
- Names should be wisely chosen and adhere to the camelCase convention.
- Code duplication should be avoided.

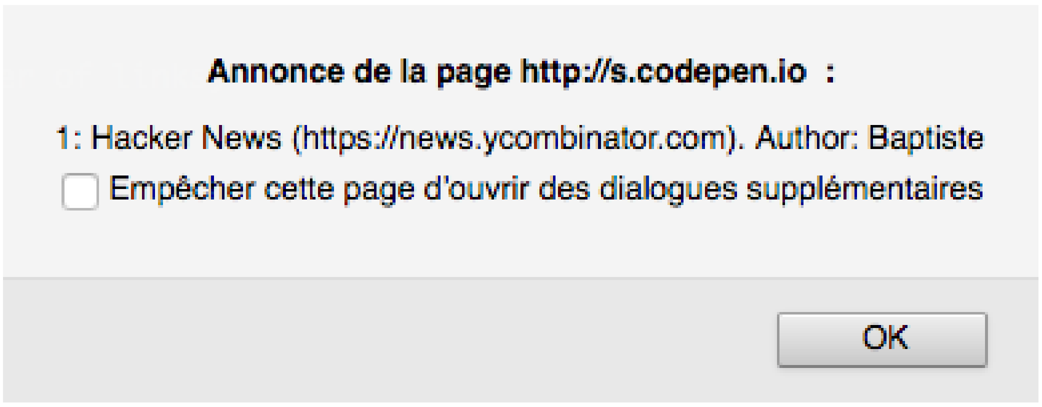
Expected result

Here are a few screenshots of the expected result.



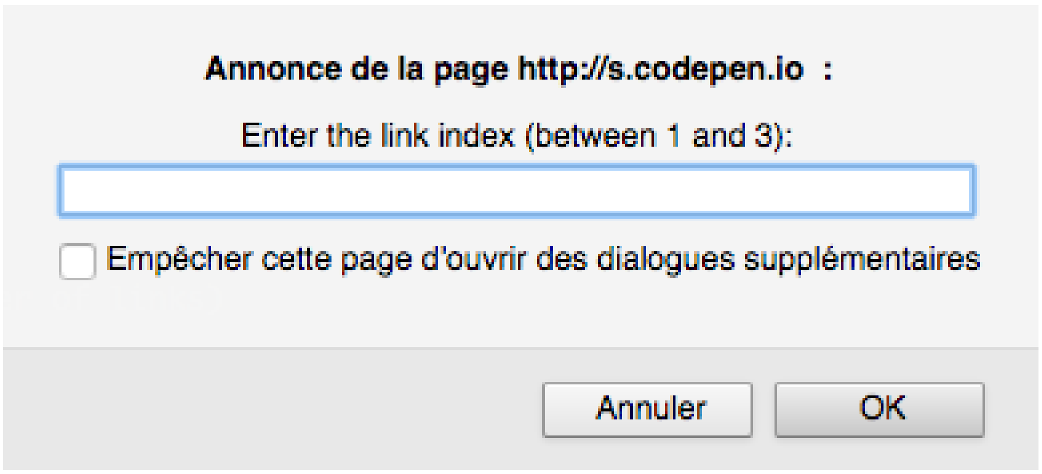
```
Annonce de la page http://s.codepen.io :

Choose an option:
1 : Show links
2 : Add a link
3 : Remove a link
0 : Quit
```

Start menu



```
Annonce de la page http://s.codepen.io :

1: Hacker News (https://news.ycombinator.com). Author: Baptiste
```

Showing a link


Annonce de la page http://s.codepen.io :

Enter the link index (between 1 and 3):

Selecting a link index

II Create interactive web pages

12. What’s a web page?

This short chapter summarizes what you need to known about the Web and web pages.

TL;DR

- The World Wide Web[1] (or **Web**) is an information space built on top of the Internet[2]. Web resources are accessible via their URL[3], and can contain hyperlinks[4] to other resources.
- A **web page** is a document suitable for the Web. Creating web pages usually involves three technologies: HTML[5] to structure the content, CSS[6] to define its presentation and JavaScript to add interactivity.
- An HTML document is made of text and structural elements called **tags** that describe the page content: paragraphs, headings, hyperlinks, images, etc.
- CSS uses **selectors** to declare which HTML elements a style applies to. Elements can be selected by tag name (`h1`), by class (`.done`) or by identifier (`#rude`).
- An HTML document can include a CSS stylesheet with a `<link>` tag and a JavaScript file with a `<script>` tag.

```
<!doctype html>
<html>

<head>
    <!-- Info about the page: title, character set, etc -->

    <!-- Link to a CSS stylesheet -->
    <link href="path/to/file.css" rel="stylesheet" type="text/css">
</head>

<body>
    <!-- Page content -->

    <!-- Link to a JavaScript file -->
    <script src="path/to/file.js"></script>
</body>

</html>
```

[1]https://en.wikipedia.org/wiki/World_Wide_Web
[2]https://en.wikipedia.org/wiki/Internet
[3]https://en.wikipedia.org/wiki/Uniform_Resource_Locator
[4]https://en.wikipedia.org/wiki/Hyperlink
[5]https://en.wikipedia.org/wiki/HTML
[6]https://en.wikipedia.org/wiki/Cascading_Style_Sheets

- A **browser** is the software you use to visit webpages and use web applications. The modern ones include a set of **developer tools** to ease the task of developing for the web.

Internet and the Web

As you probably know, the World Wide Web[7] (or **Web** for short) is an ever-expanding information space built on top of the Internet[8]. Web resources are accessible via their address, called their URL[9], and can contain hyperlinks[10] to other resources. Together, all these interlinked resources form a huge mesh analogous to a spider web.

Documents suitable for the Web are called **web pages**. They are grouped together on **websites** and visited through a special kind of software called a browser[11].

The languages of the Web

There are three main technologies for creating web pages: HTML, CSS and JavaScript.

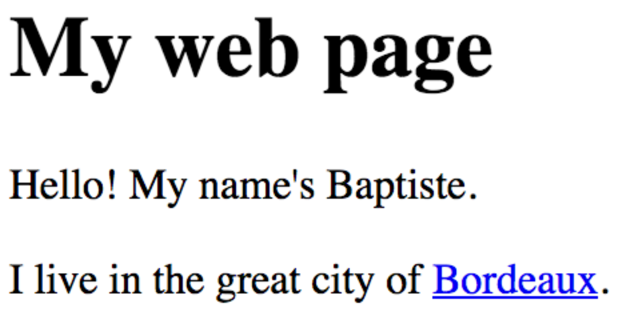
HTML

HTML, short for HyperText Markup Language[12], is the document format of web pages. An HTML document is made of text and structural elements called **tags**. Tags are used to describe the page content: paragraphs, headings, hyperlinks, images, etc.

Here is an example of a simple web page, usually stored as an `.html` file.

```
<!doctype html>
<html>

<head>
    <meta charset="utf-8">
    <title>My web page</title>
</head>

<body>
    <h1>My web page</h1>
    <p>Hello! My name's Baptiste.</p>
    <p>I live in the great city of <a href="https://en.wikipedia.org/wiki/Bor\
deaux">Bordeaux</a>.</p>
</body>

</html>
```

[7] https://en.wikipedia.org/wiki/World_Wide_Web
[8] https://en.wikipedia.org/wiki/Internet
[9] https://en.wikipedia.org/wiki/Uniform_Resource_Locator
[10] https://en.wikipedia.org/wiki/Hyperlink
[11] https://en.wikipedia.org/wiki/Web_browser
[12] https://en.wikipedia.org/wiki/HTML



My web page

Hello! My name's Baptiste.

I live in the great city of Bordeaux.

Display result

Here are a few references for learning more about HTML:

- Khan Academy - Intro to HTML[13]
- Mozilla Developer Network - HTML reference[14]

CSS

CSS, or Cascading Style Sheets[15], is a language used to alter the presentation of web pages.

CSS uses **selectors** to declare which HTML elements a style applies to. Many selecting strategies are possible, most notably:

- All elements of a given tag name.
- Elements matching a given **class** (selector syntax: `.myClass`).
- The element matching a given and unique **identifier** (selector syntax: `#MyId`).

Here is an example of a simple CSS style sheet, usually stored as a `.css` file.

```
/* All h1 elements are pink */
h1 {
   color: pink;
}

/* All elements with the class "done" are strike through */
.done {
  text-decoration: line-through;
}

/* The element having id "rude" is shown uppercase with a particular font */
#rude {
  font-family: monospace;
  text-transform: uppercase;
}
```

[13] https://www.khanacademy.org/computing/computer-programming/html-css#intro-to-html
[14] https://developer.mozilla.org/en-US/docs/Web/HTML/Reference
[15] https://en.wikipedia.org/wiki/Cascading_Style_Sheets

A style sheet is associated with an HTML document using a `link` tag in the `head` part of the page.

```html
<!-- Link to a CSS stylesheet -->
<link href="path/to/file.css" rel="stylesheet" type="text/css">
```

To learn more about CSS, visit the following links:

- Khan Academy - Intro to CSS[16]
- Mozilla Developer Network - CSS reference[17]

JavaScript

JavaScript can interact with an HTML document to provide dynamic interactivity: responses to user actions on the page, dynamic styling, animations, etc. It is the only programming language understood by all web browsers.

A JavaScript file, usually stored in a `.js` file, is loaded by a web page with a `<script>` tag.

```html
<!-- Link to a JavaScript file -->
<script src="path/to/file.js"></script>
```

Developing web pages

To create interactive web pages, you need to write HTML, CSS and JavaScript code. If you’re just starting out, the easiest way to do so is by using an online JavaScript playground like CodePen[18]. However, you will likely want to develop in a more professional fashion at some point. You may also need to work offline, which is not supported by CodePen.

Using a code editor

To do so, you’ll need a **code editor**, a software that will assist you during coding. Here are some of them:

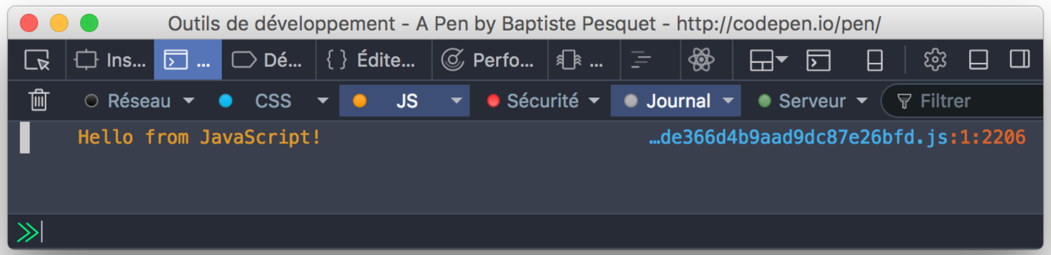
- Visual Studio Code[19] (my personal favorite).
- Brackets[20].
- Atom[21].
- Sublime Text[22].

[16]https://www.khanacademy.org/computing/computer-programming/html-css#intro-to-css
[17]https://developer.mozilla.org/en-US/docs/Web/CSS/Reference
[18]http://codepen.io
[19]https://code.visualstudio.com/
[20]http://brackets.io/
[21]https://atom.io/
[22]https://www.sublimetext.com/

Browser and developer tools

A **browser** is the software you use to visit webpages and use web applications. Modern browser include **developer tools** to help with web development. Each browser differs in exactly which tools they provide, but they're more similarities than differences among them.

These tools usually include a **JavaScript console** (to show JS output and type commands), a **page inspector** (to browse the page structure) and many more!


Outils de développement - A Pen by Baptiste Pesquet - http://codepen.io/pen/
Ins…
…
Dé…
Édite…
Perfo…
Réseau
CSS
JS
Sécurité
Journal
Serveur
Filtrer

```
Hello from JavaScript!                …de366d4b9aad9dc87e26bfd.js:1:2206
```

The Firefox JavaScript console

Check out the following links to discover more about browser developer tools:

- Khan Academy - Inspecting HTML and CSS[23].
- OpenClassrooms - Optimize your website with DevTools[24].
- Chrome DevTools Overview[25].
- Firefox Developer Tools[26].

> Even if you're using CodePen to follow along this book, you can still use the developer tools in addition to the CodePen console. For performance reasons, the CodePen console does not always show the same amount of information as the "real" browser console.

Coding time!

You can skip this exercise if you ahve prior experience with HTML and CSS.

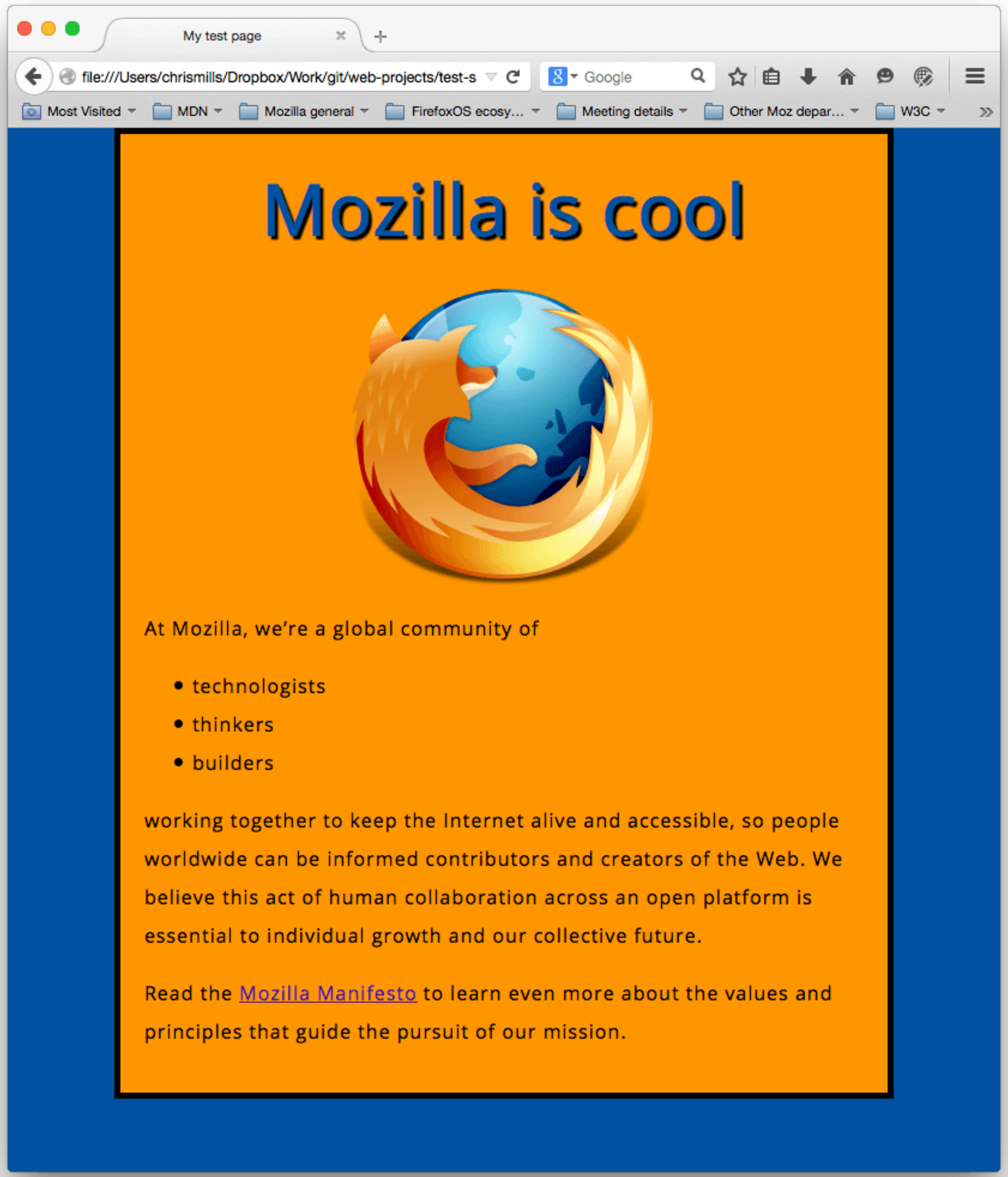
[23]https://www.khanacademy.org/computing/computer-programming/html-css/web-development-tools/a/using-the-browser-developer-tools
[24]https://openclassrooms.com/courses/optimize-your-website-with-devtools
[25]https://developer.chrome.com/devtools
[26]https://developer.mozilla.org/son/docs/Tools

Your first web page

Using CodePen or working offline, follow the beginning of the Getting started with the Web[27] tutorial from Mozilla Develper Network to create a simple web page using HTML and CSS. The required steps are:

1. What will your website look like?[28]
2. Dealing with files[29]
3. HTML basics[30]
4. CSS basics[31]

[27]https://developer.mozilla.org/en-US/docs/Learn/Getting_started_with_the_web
[28]https://developer.mozilla.org/en-US/docs/Learn/Getting_started_with_the_web/What_will_your_website_look_like
[29]https://developer.mozilla.org/en-US/docs/Learn/Getting_started_with_the_web/Dealing_with_files
[30]https://developer.mozilla.org/en-US/docs/Learn/Getting_started_with_the_web/HTML_basics
[31]https://developer.mozilla.org/en-US/docs/Learn/Getting_started_with_the_web/CSS_basics



Mozilla is cool

At Mozilla, we're a global community of

- technologists
- thinkers
- builders

working together to keep the Internet alive and accessible, so people worldwide can be informed contributors and creators of the Web. We believe this act of human collaboration across an open platform is essential to individual growth and our collective future.

Read the Mozilla Manifesto to learn even more about the values and principles that guide the pursuit of our mission.

Expected result

13. Discover the DOM

This chapter will help you discover how a web page is shown by a browser.

TL;DR

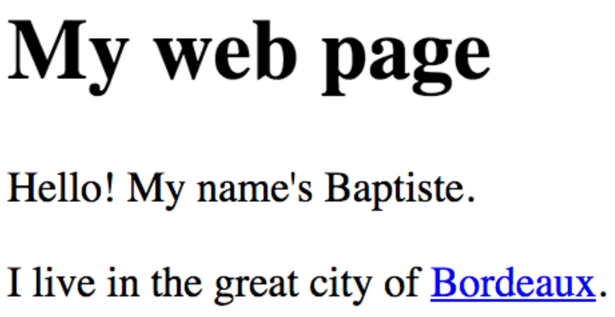
- A **web page** is a structured document containing both text and HTML tags. The **DOM**, or *Document Object Model*, is a standardized way to define a web page's structure.
- Tho DOM is alos an **API** allowing programmatical interactions with the web page. With JavaScript, you can access the structure of a page displayed in a browser and modify it.
- The DOM represents a web page as a hierarchy of **objects**, where each object corresponds to a node in the nested HTML element tree.
- The `document` variable provides access to the root of the DOM tree and corresponds to the `<html>` element in the HTML itself.
- DOM objects have **properties** and **methods** that you can manipulate with JavaScript. For example, `nodeType` returns the node type, `childNodes` contains a collection of child nodes, and `parentNode` returns the parent node.

Introduction to the DOM

You already know that a web page is a document that contains text and tags such as headings, paragraphs, links, etc. This happens in a language called **HTML**.

Let's take this simple web page as an example. Feel free to add your own information!

```html
<!doctype html>
<html>

<head>
    <meta charset="utf-8">
    <title>My web page</title>
</head>

<body>
    <h1>My web page</h1>
    <p>Hello! My name's Baptiste.</p>
    <p>I live in the great city of <a href="https://en.wikipedia.org/wiki/Bor\
deaux">Bordeaux</a>.</p>
</body>

</html>
```


My web page

Hello! My name's Baptiste.

I live in the great city of Bordeaux.

Display result

To create this result, the browser first takes the HTML code and builds a representation of its structure. It then displays this structure in the browser.

The browser also offers a *programmatical* access to its structured representation of a displayed web page. Using this interface, you can dynamically update the page: adding or removing elements, changing styles, etc. This is how you create **interactive** web pages.

The structured representation of a web page is called **DOM**, short for *Document Object Model*. The DOM defines the structure of a page and a way to interact with it. This means it's a programming interface, or **API** (*Application Programming Interface*). JavaScript is the language of choice for interacting with the DOM.

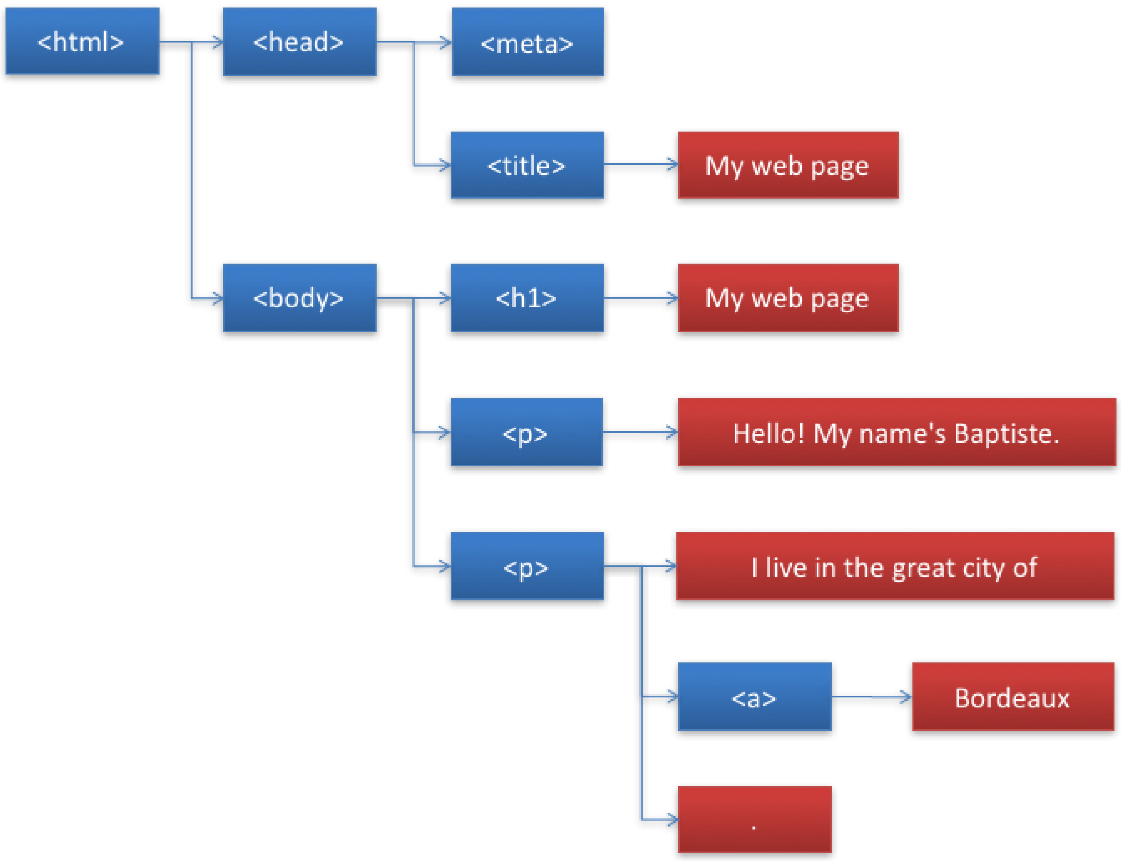
> At the dawn of the Web, each browser was using its own DOM, giving headaches to JavaScript developers trying to code web pages. These hard times are over: through a World Wide Web Consortium[1] (W3C) effort, the first version of a unified DOM was created in 1998. Nowadays, all recent browsers use a standardized DOM.

Web page structure

A web page is a set of nested tags. You can represent it in a hierarchical form called a **tree**. The `<html>` element sets up your document as HTML and contains two sub-elements, `<head>` and `<body>`, which themselves contain several sub-elements.

Here is the tree corresponding to our example HTML page.

[1]https://w3c.org



Example structure

Each entity in the tree is called a **node**. There are two types of nodes:

- Those (in blue here) that correspond to HTML tags like `<body>` or `<p>`. These nodes are called **element nodes** and they can have subnodes, called **child nodes** or children.
- Those (in red) that match the textual content of the page. These nodes are called **text nodes** and do not have children.

Get started with the DOM in JavaScript

The DOM represents a web page as a hierarchy of objects, where each object corresponds to a node in the nested HTML element tree. DOM objects have **properties** and **methods** that you can manipulate with JavaScript.

Access the DOM with the `document` variable

When a JavaScript program runs in the context of a web browser, it can access the root of the DOM using the variable `document`. This variable matches the `<html>` element.

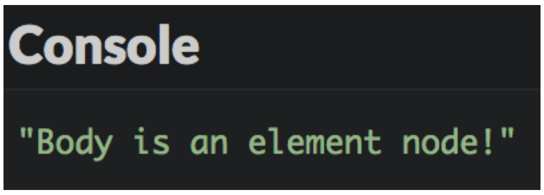
`document` is an object that has `head` and `body` properties which allow access to the `<head>` and `<body>` elements of the page.

```
const h = document.head; // "h" variable contains the contents of the DOM's h\
ead
const b = document.body; // "b" variable contains the contents of the DOM's b\
ody
```

Discover a node's type

Each object has a property called `nodeType` which indicates its type. The value of this property is `document.ELEMENT_NODE` for an "element" node (otherwise known as an HTML tag) and `document.TEXT_NODE` for a text node.

```
if (document.body.nodeType === document.ELEMENT_NODE) {
    console.log("Body is an element node!");
} else {
    console.log("Body is a textual node!");
}
```



```
Console
"Body is an element node!"
```

Execution result

As expected, the DOM object `body` is an element node because it's an HTML tag.

Access a node's children

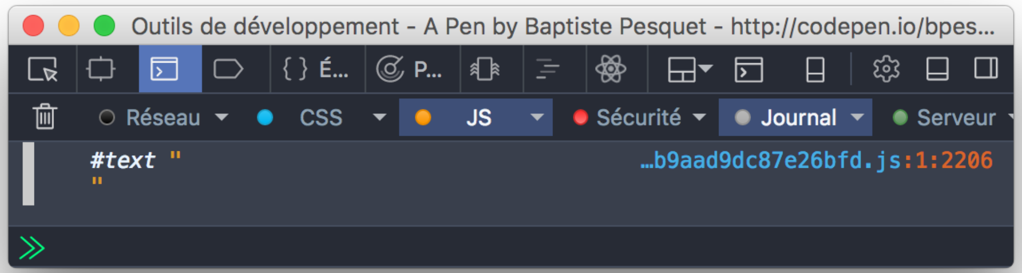
Each element-typed object in the DOM has a property called `childNodes`. This is an ordered collection containing all its child nodes as DOM objects. You can use this collection a bit like an array to access the different children of a node.

The `childNodes` property of an element node is not a real JavaScript array, but rather a NodeList[2] object. Not all of the standard array methods are applicable to it.

The following code would display the first child of the `body` node.

```
// Access the first child of the body node
console.log(document.body.childNodes[0]);
```

[2]https://developer.mozilla.org/en-US/docs/Web/API/NodeList


Outils de développement - A Pen by Baptiste Pesquet - http://codepen.io/bpes...
Réseau
CSS
JS
Sécurité
Journal
Serveur

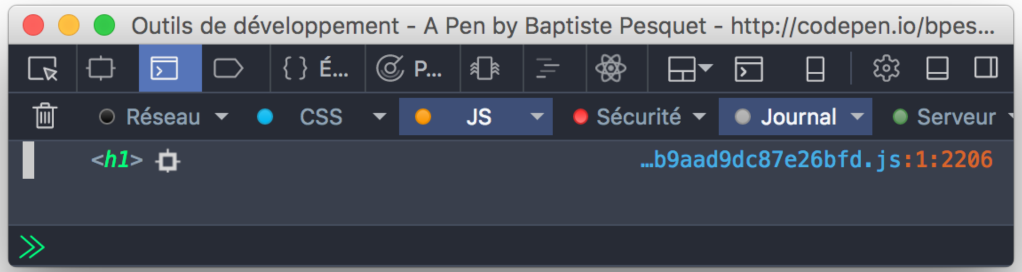
```
#text "
"                                    …b9aad9dc87e26bfd.js:1:2206
```

Execution result

Wait… Why isn't the first child node `h1`, since that's the first element in the body's HTML?

That's because spaces between tags and line returns in HTML code are considered text nodes by the browser. The node `h1` is therefore the *second* child node of the body. Let's double check that:

```
// Access the second child of the body node
console.log(document.body.childNodes[1]);
```


Outils de développement - A Pen by Baptiste Pesquet - http://codepen.io/bpes...
Réseau
CSS
JS
Sécurité
Journal
Serveur

```
<h1>                                 …b9aad9dc87e26bfd.js:1:2206
```

Execution result

To eliminate these text nodes between tags, you could have written the HTML page in a more condensed way.

```
<body><h1>My web page</h1><!-- ... -->
```

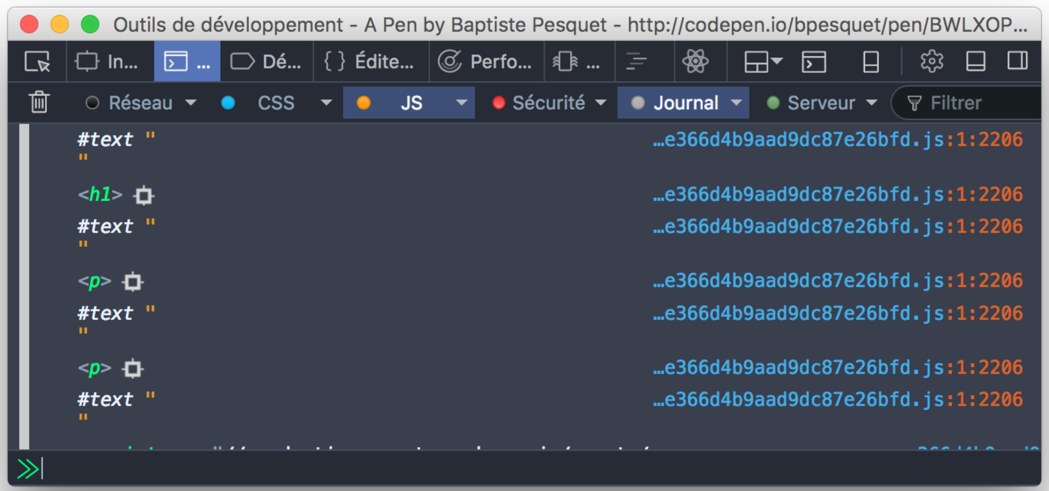
It's better, however, to take the text nodes between tags into accouant than to sacrifice lisibility and code indentation.

Browse child nodes

To browse a list of child nodes, you can use a classical `for` loop, the `forEach()` method or the newer `for-of` loop as seen below:

```
// Browse the body node's children using a for loop@
for (let i = 0; i < document.body.childNodes.length; i++) {
    console.log(document.body.childNodes[i]);
}

// Browse the body node's children using the forEach() method
document.body.childNodes.forEach(node => {
    console.log(node);
});

// Browse the body node's children using a for-of loop
for (const node of document.body.childNodes) {
    console.log(node);
}
```

Each of these techniques gives the following result.


Outils de développement - A Pen by Baptiste Pesquet - http://codepen.io/bpesquet/pen/BWLXOP...
In...
...
Dé...
Édite...
Perfo...
...
Réseau
CSS
JS
Sécurité
Journal
Serveur
Filtrer

```
#text "
"                                      …e366d4b9aad9dc87e26bfd.js:1:2206
<h1>                                   …e366d4b9aad9dc87e26bfd.js:1:2206
#text "                                …e366d4b9aad9dc87e26bfd.js:1:2206
"
<p>                                    …e366d4b9aad9dc87e26bfd.js:1:2206
#text "                                …e366d4b9aad9dc87e26bfd.js:1:2206
"
<p>                                    …e366d4b9aad9dc87e26bfd.js:1:2206
#text "                                …e366d4b9aad9dc87e26bfd.js:1:2206
"
```

Execution result

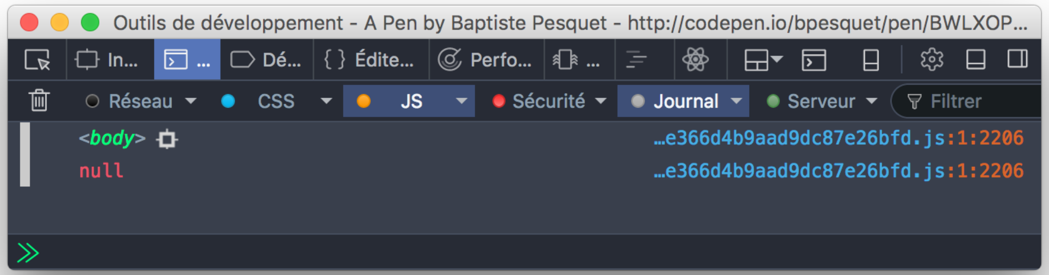
Again, spaces and line returns count as text nodes in the DOM.

Access a node's parent

Each DOM object has a property called `parentNode` that returns its parent node as a DOM object.

For the DOM root node (`document`), the value of `parentNode` is `null` since it has no parent node.

```
const h1 = document.body.childNodes[1];
console.log(h1.parentNode);         // Show the body node
console.log(document.parentNode); // Will show null, since body has no parent\
 node
```


Outils de développement - A Pen by Baptiste Pesquet - http://codepen.io/bpesquet/pen/BWLXOP...
Réseau
CSS
JS
Sécurité
Journal
Serveur
Filtrer

```
<body>                                   ...e366d4b9aad9dc87e26bfd.js:1:2206
null                                     ...e366d4b9aad9dc87e26bfd.js:1:2206
```

Execution result

There are other properties that we will not discuss here that let you navigate through the DOM, like `firstChild`, `lastChild` or `nextSibling`.

Coding time!

Showing a node's child

Your mission here is to create a `showChild()` function that shows one a of children of an DOM element node. This function takes as parameter the parent node and the child node index. Error cases like non-element node or out-of-limits index must be taken into account.

Here'e the associated HTML code.

```
<h1>A title</h1>
<div>Some text with <a href="#">a link</a>.</div>
```

Complete the following program to obtain the expected results.

```
// Show a DOM object's child node
// "node" is the DOM object
// "index" is the index of the child node
function showChild(node, index) {
    // TODO: add code here
}

// Should show the h1 node
showChild(document.body, 1);

// Should show "Incorrect index"
showChild(document.body, -1);

// Should show "Incorrect index"
showChild(document.body, 8);

// Should show "Wrong node type"
showChild(document.body.childNodes[0], 0);
```

Use `console.error()` rather than `console.log()` to display an error message in the console.

14. Traverse the DOM

In this chapter, you'll see how to use JavaScript to traverse the DOM.

TL;DR

- Rather than go through the DOM node by node, you can quickly access one or more elements using **selection methods**.
- The `getElementsByTagName()`, `getElementsByClassName()` and `getElementById()` methods respectively search items by **tag name**, **class**, and **ID**. The first two methods return a list, and the latter returns a single item.
- The `querySelectorAll()` and `querySelector()` methods make it possible to search for items using a **CSS selector**. The first method returns all matching items, and the second returns only the first.
- The `innerHTML` property returns the **HTML content** of an element. The `textContent` property returns its **textual content** without any HTML markup.
- The `getAttribute()` and `hasAttribute()` methods allow access to element **attributes**. The `classList` property and its method `contains()` provides access to an element's **classes**.

Sample web page

Here's the example web page used throughout this chapter.

```
<h1>Seven wonders of the world</h1>
<p>Do you know the seven wonders of the world?</p>
<div id="content">
    <h2>Wonders from Antiquity</h2>
    <p>This list comes to us from ancient times.</p>
    <ul class="wonders" id="ancient">
        <li class="exists">Great Pyramid of Giza</li>
        <li>Hanging Gardens of Babylon</li>
        <li>Lighthouse of Alexandria</li>
        <li>Statue of Zeus at Olympia</li>
        <li>Temple of Artemis at Ephesus</li>
        <li>Mausoleum at Halicarnassus</li>
        <li>Colossus of Rhodes</li>
    </ul>
    <h2>Modern wonders of the world</h2>
    <p>This list was decided by vote.</p>
    <ul class="wonders" id="new">
        <li class="exists">Petra</li>
        <li class="exists">Great Wall of China</li>
        <Li class="exists">Christ the Redeemer</Li>
        <Li class="exists">Machu Picchu</Li>
        <li class="exists">Chichen Itza</li>
        <li class="exists">Colosseum</li>
        <li class="exists">Taj Mahal</li>
    </ul>
    <h2>References</h2>
    <ul>
        <li><a href="https://en.wikipedia.org/wiki/Seven_Wonders_of_the_Ancie\
nt_World">Seven Wonders of the Ancient World</a></li>
        <li><a href="https://en.wikipedia.org/wiki/New7Wonders_of_the_World">\
New Wonders of the World</a></li>
    </ul>
</div>
```

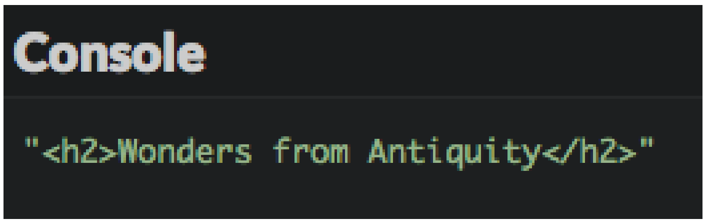
Selecting elements

The limits of node-by-node traversal

In the previous chapter, you saw how to navigate the DOM node structure of a web page beginning with the root node and using the `childNodes` property to move down levels in the structure of the page.

Suppose you want to select the title `"Wonders from Antiquity"` of our web page. Taking into account the text nodes between elements, this node is the second child node of the sixth child node of the `body` element. So you could write something like this.

```
// Show the "Wonders from Antiquity" h2 element
console.log(document.body.childNodes[5].childNodes[1]);
```



```
"<h2>Wonders from Antiquity</h2>"
```

Execution result

This technique is pretty awkward and error-prone. The code is difficult to read and must be updated if new elements are further inserted in the web page. Fortunately, there are much better solutions.

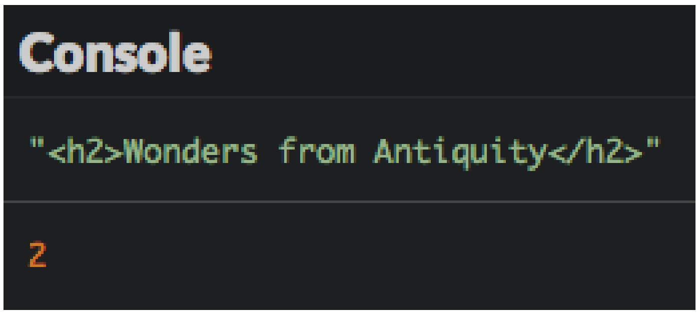
Selecting items according to HTML tag

All DOM elements have a method called `getElementsByTagName()`. This returns, under the form of a NodeList[1], a list of items that have the name of the tag that's passed as a parameter.

The search happens through all the sub-elements of the node on which the method is called – not only its direct children.

With this method, selecting the first `h2` element becomes super easy:

```
// Get all h2 elements
const titleElements = document.getElementsByTagName("h2");

console.log(titleElements[0]);      // Show the first h2
console.log(titleElements.length); // 2 (total number of h2 elements in the p\
age)
```


```
Console
"<h2>Wonders from Antiquity</h2>"
2
```

Execution result

Suffixing JavaScript variables associated to DOM element nodes with `Element` (or `Elements` when the variable contains several nodes) is a popular naming convention. We'll stick to it throughout this book.

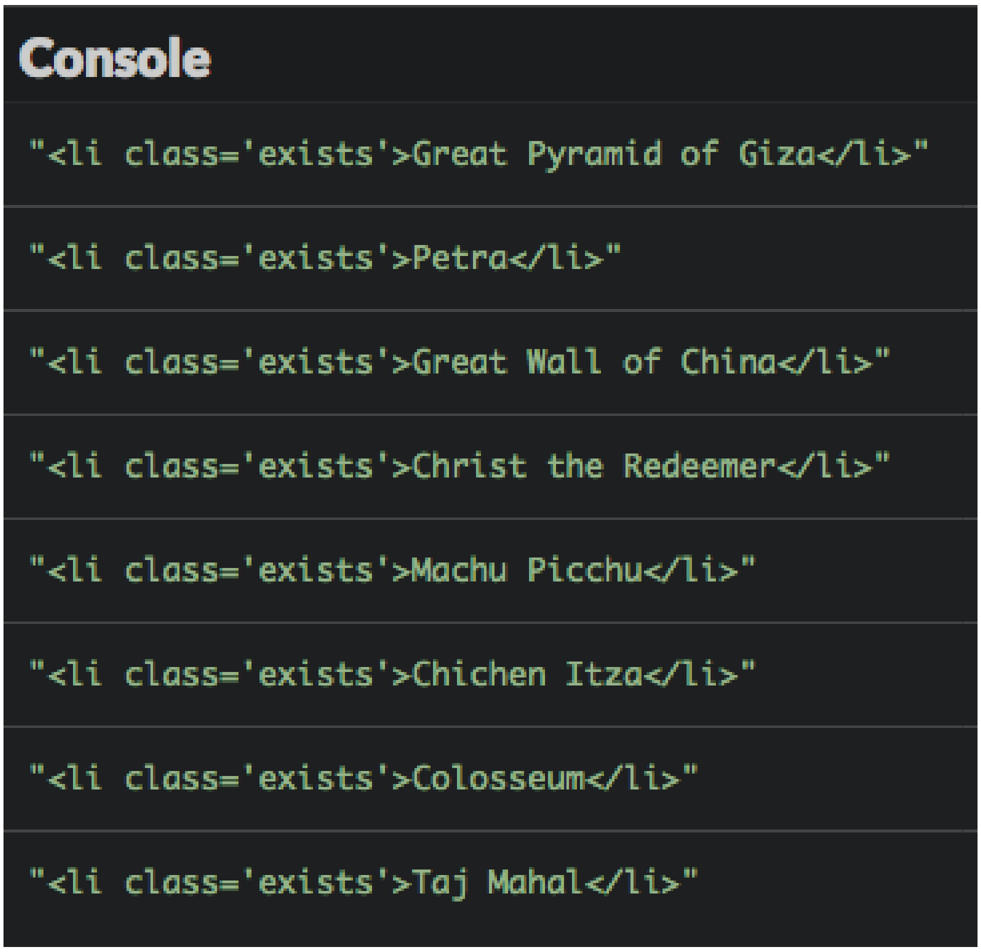
Selecting items according to class

DOM elements also feature a method called `getElementsByClassName()`. It returns a list of elements with the class name as a parameter. Again, the search covers all sub-elements of the node on which the method is called.

To select and display all document elements with a class `"wonders"`, you can write the following code.

[1] https://developer.mozilla.org/en-US/docs/Web/API/NodeList

```
// Show all elements that have the class "exists"
const existingElements = document.getElementsByClassName("exists");
for (const element of existingElements) {
    console.log(element);
}
```



Console

```
"<li class='exists'>Great Pyramid of Giza</li>"
"<li class='exists'>Petra</li>"
"<li class='exists'>Great Wall of China</li>"
"<li class='exists'>Christ the Redeemer</li>"
"<li class='exists'>Machu Picchu</li>"
"<li class='exists'>Chichen Itza</li>"
"<li class='exists'>Colosseum</li>"
"<li class='exists'>Taj Mahal</li>"
```

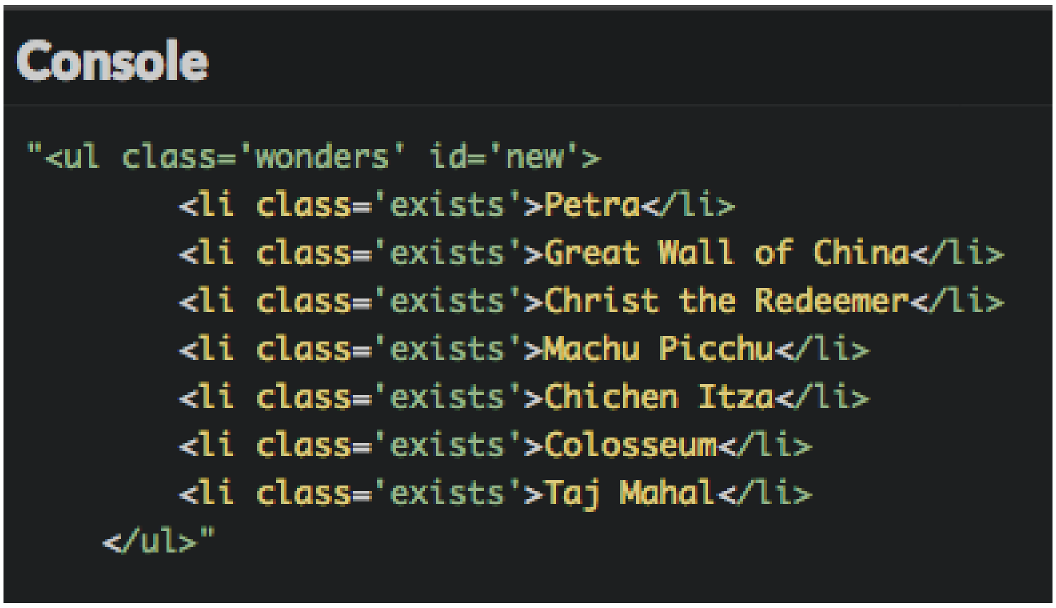
Execution result

Selecting an item according to its ID

Lastly, each element of the DOM provides a method called `getElementById()` that returns among all sub-elements with the ID passed as a parameter. It returns `null` if no associated element can be found.

The following code selects and displays the list with ID `"new"`.

```
// Show element with the ID "new"
console.log(document.getElementById("new"));
```



```
Console

"<ul class='wonders' id='new'>
        <li class='exists'>Petra</li>
        <li class='exists'>Great Wall of China</li>
        <li class='exists'>Christ the Redeemer</li>
        <li class='exists'>Machu Picchu</li>
        <li class='exists'>Chichen Itza</li>
        <li class='exists'>Colosseum</li>
        <li class='exists'>Taj Mahal</li>
    </ul>"
```

Execution result

Beware: contrary to others, the `getElementById()` method does not contain any `'s'` after the `"Element"` word.

Selecting elements via CSS selectors

For more complex use cases, you can also use CSS selectors to access DOM elements.

For example, let's say that you want to grab all the `<li>` elements of wonders that are both ancient and still exist.

```
// All "ancient" wonders that still exist
console.log(document.getElementById("ancient").getElementsByClassName("exists\
").length); // 1
```

This syntax is a little clunky though. Let's learn two new methods that make finding elements easier.

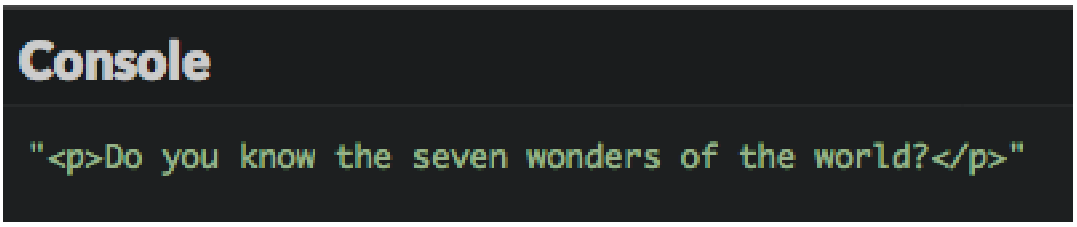
The first is `querySelectorAll()`, with which you can use CSS selectors to identify elements.

```
// All paragraphs
console.log(document.querySelectorAll("p").length); // 3

// All paragraphs inside the "content" ID block
console.log(document.querySelectorAll("#content p").length); // 2

// All elements with the "exists" class
console.log(document.querySelectorAll(".exists").length); // 8

// All "ancient" wonders that still exist
console.log(document.querySelectorAll("#ancient > .exists").length); // 1
```

> Check the Mozille Developer Network[2] for a primer on the different CSS selectors available.

The second method using CSS selectors is called `querySelector()`. It works the same way as `querySelectorAll()` but only returns the first matching element. It returns `null` if no associated element can be found.

```
// Show the first paragraph
console.log(document.querySelector("p"));
```



```
Console
"<p>Do you know the seven wonders of the world?</p>"
```

Execution result

Choosing a selection method

You just discovered several ways of selecting DOM elements. How to choose the right one?

Since they use CSS selectors, `querySelectorAll()` and `querySelector()` could cover all your needs, but they might perform slower[3] than the others.

Here are the general rules of thumb that you should follow.

Number of items to get	Selection criterion	Method to use
Many	By tag	`getElementsByTagName()`
Many	By class	`getElementsByClassName()`
Many	Not by class or tag	`querySelectorAll()`
Only one	By ID	`getElementById()`
Only one (the first)	Not by ID	`querySelector()`

[2]https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Selectors
[3]https://jsperf.com/getelementsbyclassname-vs-queryselectorall/195

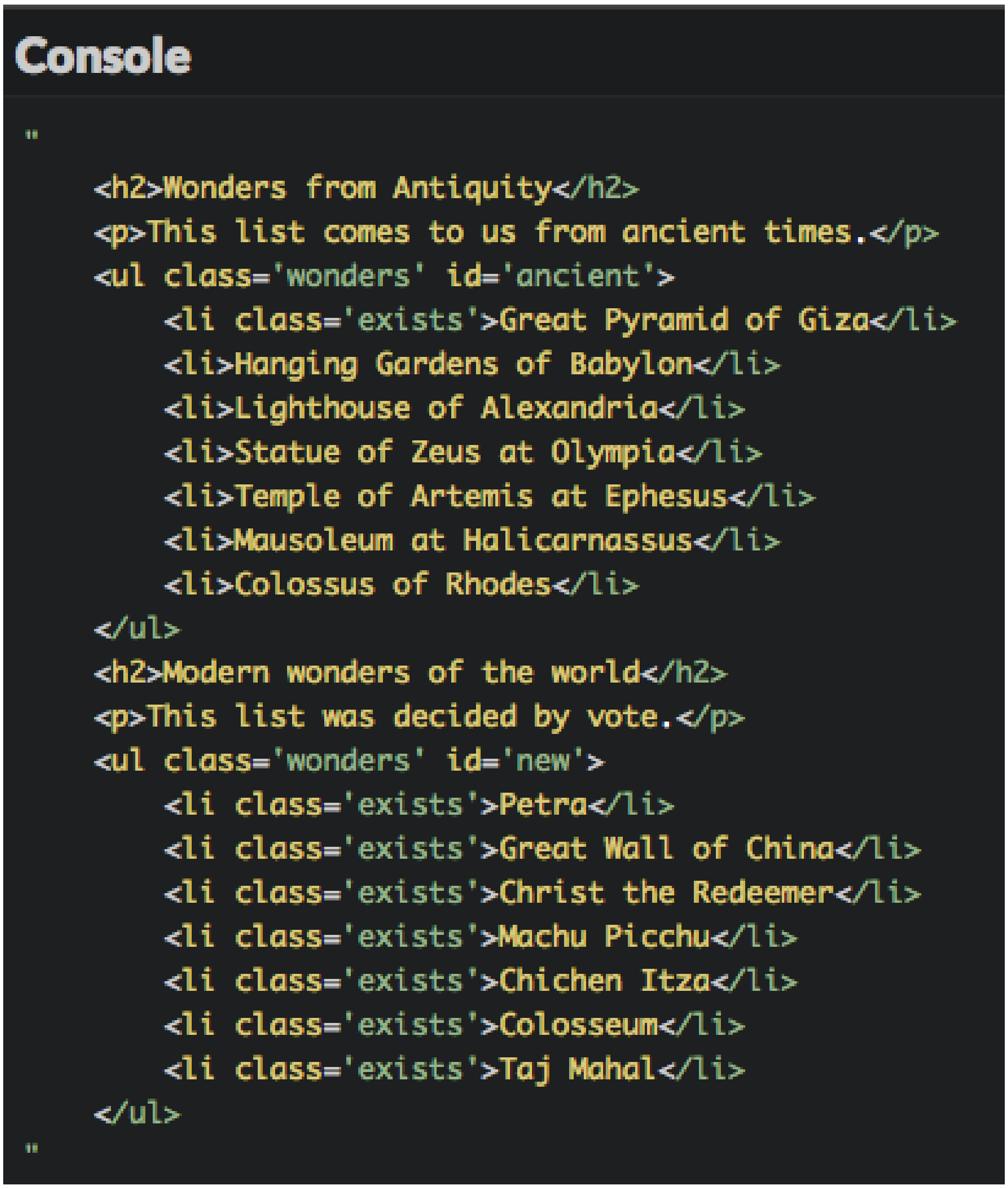
Obtaining information about elements

The DOM also provides information on the items you've just selected.

HTML content

The `innerHTML` property will retrieve the HTML contents of your DOM element.

```
// The HTML content of the DOM element with ID "content"
console.log(document.getElementById("content").innerHTML);
```



```
"
    <h2>Wonders from Antiquity</h2>
    <p>This list comes to us from ancient times.</p>
    <ul class='wonders' id='ancient'>
        <li class='exists'>Great Pyramid of Giza</li>
        <li>Hanging Gardens of Babylon</li>
        <li>Lighthouse of Alexandria</li>
        <li>Statue of Zeus at Olympia</li>
        <li>Temple of Artemis at Ephesus</li>
        <li>Mausoleum at Halicarnassus</li>
        <li>Colossus of Rhodes</li>
    </ul>
    <h2>Modern wonders of the world</h2>
    <p>This list was decided by vote.</p>
    <ul class='wonders' id='new'>
        <li class='exists'>Petra</li>
        <li class='exists'>Great Wall of China</li>
        <li class='exists'>Christ the Redeemer</li>
        <li class='exists'>Machu Picchu</li>
        <li class='exists'>Chichen Itza</li>
        <li class='exists'>Colosseum</li>
        <li class='exists'>Taj Mahal</li>
    </ul>
"
```

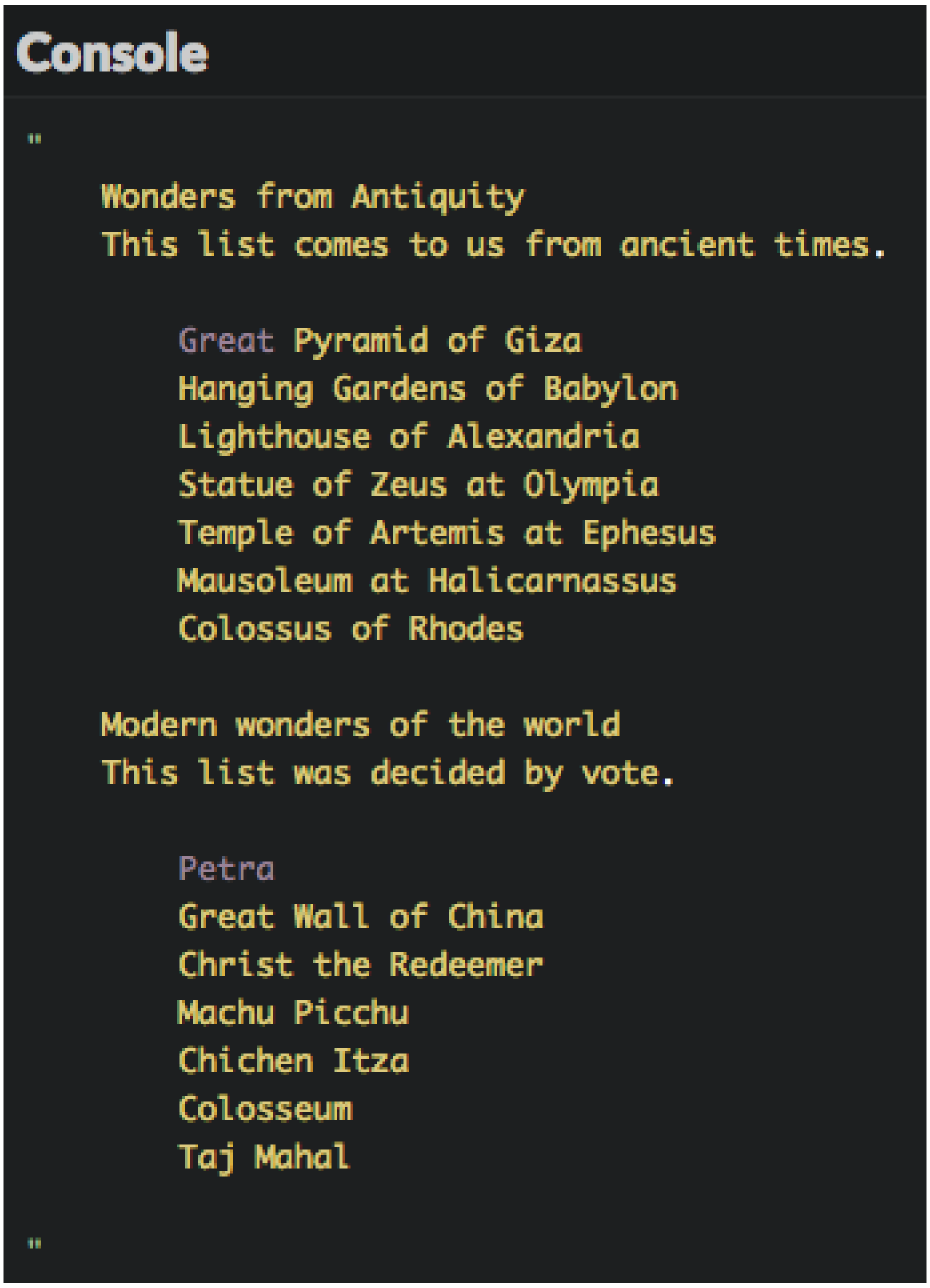
Execution result

This property has been introduced by Microsoft and is not part of the W3C DOM specification, but it is nonetheless supported by all major browsers.

Textual content

The `textContent` property returns all the text content of a DOM element, without any HTML markup.

```
// The textual content of the DOM element with ID "content"
console.log(document.getElementById("content").textContent);
```



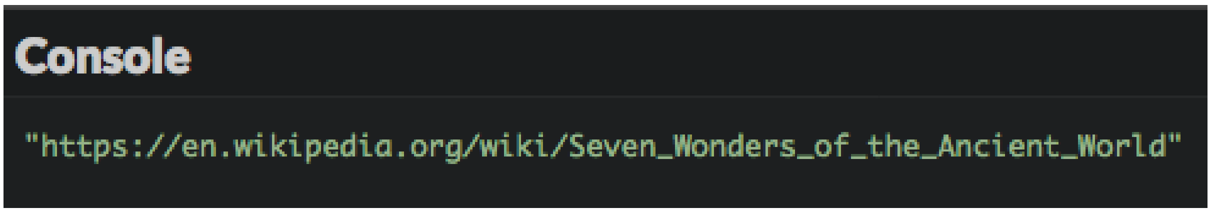
```
Console
"

    Wonders from Antiquity
    This list comes to us from ancient times.

        Great Pyramid of Giza
        Hanging Gardens of Babylon
        Lighthouse of Alexandria
        Statue of Zeus at Olympia
        Temple of Artemis at Ephesus
        Mausoleum at Halicarnassus
        Colossus of Rhodes

    Modern wonders of the world
    This list was decided by vote.

        Petra
        Great Wall of China
        Christ the Redeemer
        Machu Picchu
        Chichen Itza
        Colosseum
        Taj Mahal

"
```

Execution result

Attributes

The `getAttribute()` method can be applied to a DOM element and will return the value of a given attribute.

```
// Show href attribute of the first link
console.log(document.querySelector("a").getAttribute("href"));
```

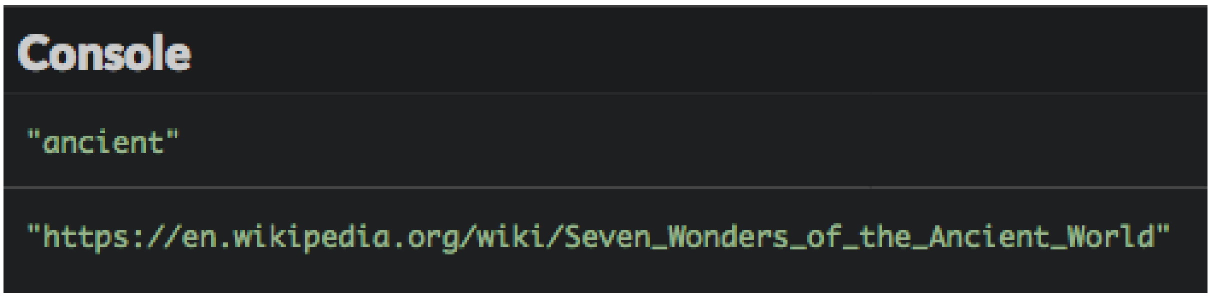


```
Console
"https://en.wikipedia.org/wiki/Seven_Wonders_of_the_Ancient_World"
```

Execution result

Some attributes are directly accessible as properties. This is true for the `id`, `href`, and `value` attributes.

```
// Show ID attribute of the first list
console.log(document.querySelector("ul").id);

// Show href attribute of the first link
console.log(document.querySelector("a").href);
```



```
Console
"ancient"
"https://en.wikipedia.org/wiki/Seven_Wonders_of_the_Ancient_World"
```

Execution result

You can check for the existence of an attribute using the `hasAttribute()` method as seen in the example below.

```
if (document.querySelector("a").hasAttribute("target")) {
    console.log("The first link has a target attribute.");
} else {
    console.log("The first link does not have a target attribute."); // Will \
be shown
}
```

Classes

In a web page, a tag can have multiple classes. The `classList` property retrieves a DOM element's list of classes.

```
// List of classes of the element identified by "ancient"
const classes = document.getElementById("ancient").classList;
console.log(classes.length); // 1 (since the element only has one class)
console.log(classes[0]);     // "wonders"
```

You also have the opportunity to test the presence of a class on an element by calling the `contains()` on the class list, passing the class to test as a parameter.

```
if (document.getElementById("ancient").classList.contains("wonders")) {
    console.log("The element with ID 'ancient' has the class 'wonders'."); //\
 Will be shown
} else {
    console.log("The element with ID 'ancient' does not have the class 'wonde\
rs'.");
}
```

This is only a part of the DOM traversal API. For more details, check the Mozilla Developer Network[4].

Coding time!

Counting elements

Here is some HTML code (content is by French poet Paul Verlaine).

```
<h1>Mon rêve familier</h1>

<p>Je fais souvent ce rêve <span class="adjective">étrange</span> et <span cl\
ass="adjective">pénétrant</span></p>
<p>D'une <span>femme <span class="adjective">inconnue</span></span>, et que j\
'aime, et qui m'aime</p>
<p>Et qui n'est, chaque fois, ni tout à fait la même</p>
<p>Ni tout à fait une autre, et m'aime et me comprend.</p>
```

Complete the following program to write the `countElements()` function, that takes a CSS selector as a parameter and returns the number of corresponding elements.

[4]https://developer.mozilla.org/en-US/docs/Web/API/Element

```
// TODO: write the countElements() function here

console.log(countElements("p"));              // Should show 4
console.log(countElements(".adjective"));     // Should show 3
console.log(countElements("p .adjective"));   // Should show 3
console.log(countElements("p > .adjective")); // Should show 2
```

Handling attributes

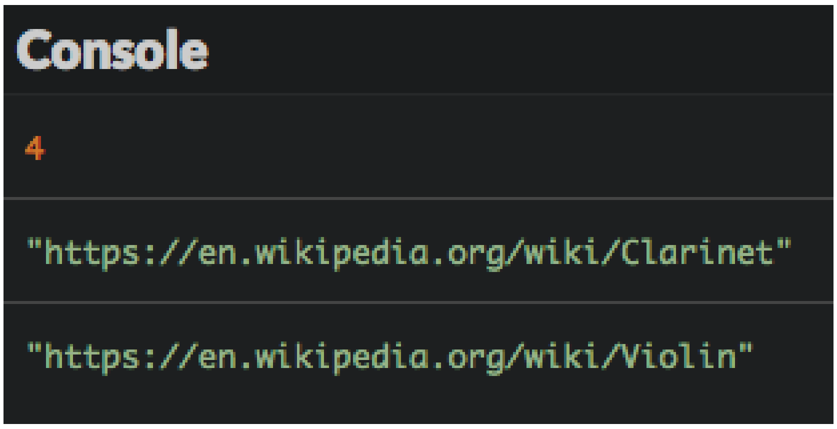
Here is the description of several musical instruments.

```
<h1>Some musical instruments</h1>
<ul>
  <li id="clarinet" class="wind woodwind">
    The <a href="https://en.wikipedia.org/wiki/Clarinet">clarinet</a>
  </li>
  <li id="saxophone" class="wind woodwind">
    The <a href="https://en.wikipedia.org/wiki/Saxophone">saxophone</a>
  </li>
  <li id="trumpet" class="wind brass">
    The <a href="https://en.wikipedia.org/wiki/Trumpet">trumpet</a>
  </li>
  <li id="violin" class="chordophone">
    The <a href="https://en.wikipedia.org/wiki/Violin">violin</a>
  </li>
</ul>
```

Write a JavaScript program containing a `linkInfo()` function that shows:

- The total number of links.
- The target of the first and last links.

This function shuld work even if no links are present.

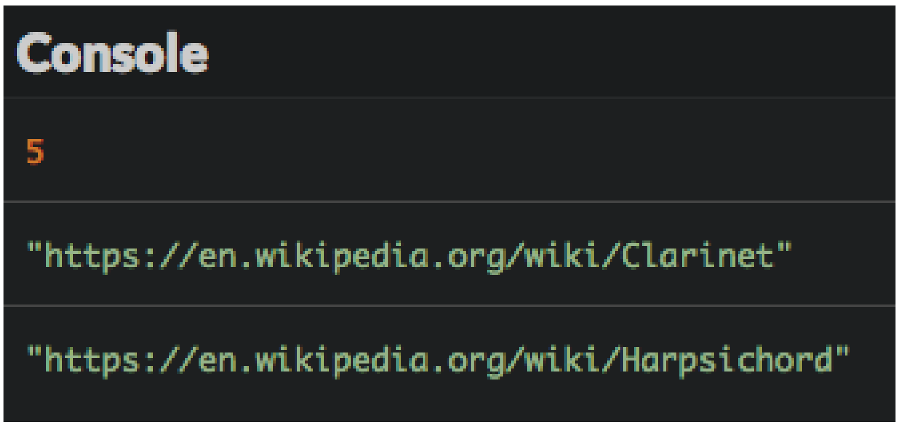

Console

```
4
"https://en.wikipedia.org/wiki/Clarinet"
"https://en.wikipedia.org/wiki/Violin"
```

Expected result

Add the following new instrument at the end of the HTML list, then check your program's new result.

```
<li id="harpsichord">
  The <a href="https://en.wikipedia.org/wiki/Harpsichord">harpsichord</a>
</li>
```



```
Console
5
"https://en.wikipedia.org/wiki/Clarinet"
"https://en.wikipedia.org/wiki/Harpsichord"
```

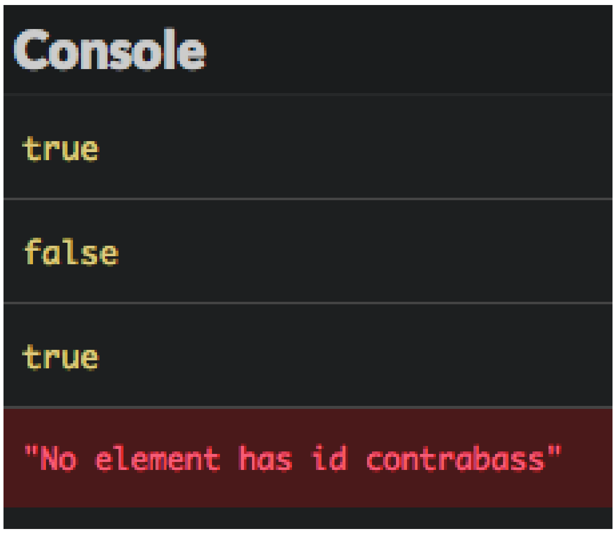
Expected result

Handling classes

Improve the previous program to add a `has()` function that test if an element designated by its ID has a class. The function shows `true`, `false` or an error message if the element can't be found.

```
// Show if an element has a class
function has(id, someClass) {
    //TODO: write the function code
}

has("saxophone", "woodwind");      // Should show true
has("saxophone", "brass");         // Should show false
has("trumpet", "brass");           // Should show true
has("contrabass", "chordophone"); // Should show an error message
```

Use `console.error()` rather than `console.log()` to display an error message in the console.



```
true
false
true
"No element has id contrabass"
```

Expected result

15. Modify the page

Let's see how to use JavaScript to modify a web page once it's been loaded by the browser! You can thus make your content more dynamic and interactive.

TL;DR

- The `innerHTML`, `textContent` and `classList` properties, as well as the `setAttribute` method, let you modify a DOM element's information.
- You create new DOM nodes via methods `createTextNode()` (for, well, text nodes) and `createElement()` (for elements themselves).
- The `appendChild()` method lets you insert a new node as the last child of a DOM element.
- The `insertBefore()` and `insertAdjacentHTML()` methods are alternative possibilities for adding content.
- You can replace existing nodes with the `replaceChild()` method or remove them with `removeChild()`.
- The JavaScript `style` property represents the `style` attribute of a DOM node. It lets you modify the element's style by defining values of its CSS properties.
- CSS properties that involve multiple words are written in **camelCase** when dealing with JavaScript. For example, `font-family` becomes `fontFamily`.
- The `style` property isn't enough to access an element's style. You should use the `getComputedStyle()` function instead.
- Manipulating the DOM with JavaScript should be done sparingly so that page performance doesn't suffer.

Modify an existing element

The DOM traversal properties studied in the previous chapter can also be used to update elements in the page.

Example page

The examples in the next paragraphs use the HTML code below.

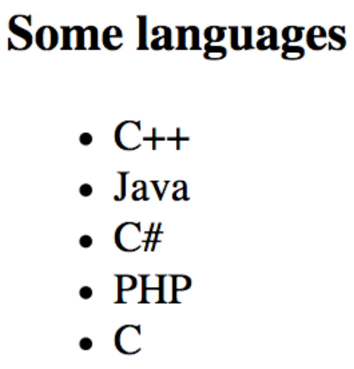
```
<h3 class="beginning">Some languages</h3>
<div id="content">
    <ul id="languages">
        <li id="cpp">C++</li>
        <li id="java">Java</li>
        <li id="csharp">C#</li>
        <li id="php">PHP</li>
    </ul>
</div>
```

HTML content

The `innerHTML` property can be used to change the content of an element within the DOM.

For example, you can add a new language to our list with the code below. We'll access the `<ul>` tag identified by `"languages"` and then add an entry to the end of the list via an operator (+=) and an `<li>`.

```
// Modifying an HTML element: adding an <li>
document.getElementById("languages").innerHTML += '<li id="c">C</li>';
```



Some languages

- C++
- Java
- C#
- PHP
- C

Execution result

The `innerHTML` property is often used to "empty" content. Try the following example:

```
// Delete the HTML content of the list, replacing it with nothing
document.getElementById("languages").innerHTML = "";
```

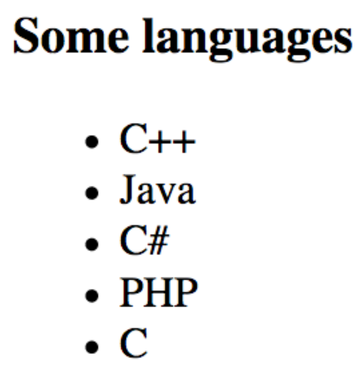
Before moving on, remove the above line from your JavaScript program. Otherwise, you'll have no content!

When using `innerHTML`, you put some HTML content into strings. To keep your code readable and avoid mistakes, you should only use `innerHTML` to make small content changes. You'll discover more versatile solutions below.

Text content

Use the `textContent` property to modify the text content of a DOM element. Here is how to complete the title displayed by our page.

```
// Modify the title's text content
document.querySelector("h3").textContent += " for programming";
```



Some languages

- C++
- Java
- C#
- PHP
- C

Execution result

Attributes

The `setAttribute()` method sets the value of an attribute of an element. You pass the name and value of the attribute as parameters.

```
// Define the id attribute of the first title
document.querySelector("h3").setAttribute("id", "title");
```

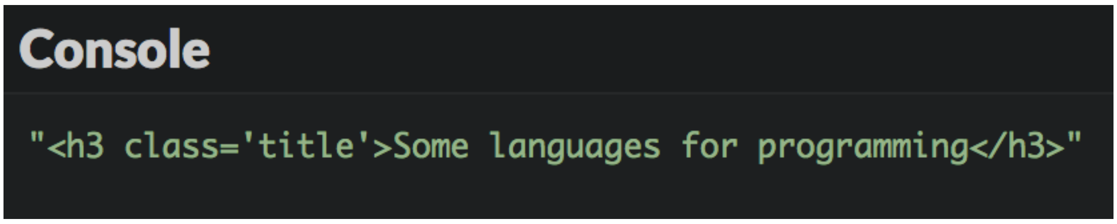
As you saw in the previous chapter, some attributes exist as properties and can be directly updated.

```
// Define the id attribute of the first title
document.querySelector("h3").id = "title";
```

Classes

You can use the `classList` property to add or remove classes from a DOM element!

```
const titleElement = document.querySelector("h3"); // Grab the first h3
titleElement.classList.remove("beginning");        // Remove the class "begin\
ning"
titleElement.classList.add("title");               // Add a class called "tit\
le"
console.log(titleElement);
```



```
"<h3 class='title'>Some languages for programming</h3>"
```

Execution result

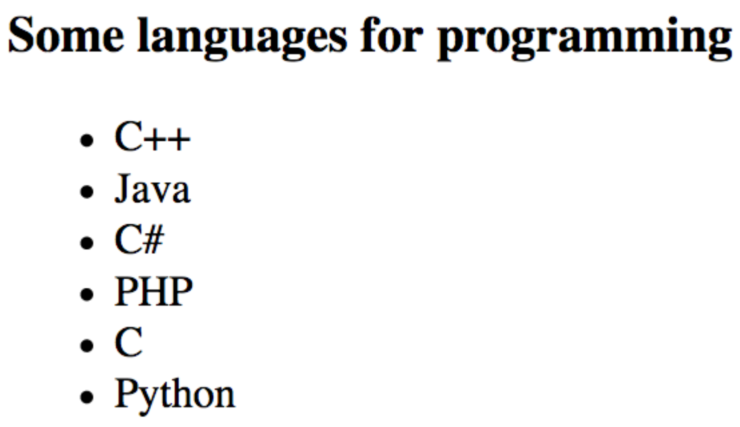
Adding a new element

Adding a new element to a web page can be broken into three steps:

- Create the new element.
- Set element properties.
- Insert the new element in the DOM.

For example, suppose you want to add the language “Python” to the list of languages on our page. Here’s the JavaScript code you’d use to do so.

```
const pythonElement = document.createElement("li"); // Create an "li" element
pythonElement.id = "python";           // Define element ID
pythonElement.textContent = "Python"; // Define its text content
document.getElementById("languages").appendChild(pythonElement); // Insert th\
e new element into the DOM
```



Some languages for programming

- C++
- Java
- C#
- PHP
- C
- Python

Execution result

Let’s study each of these steps.

Creating the element

You'd create an element using the `createElement()` method (surprising, isn't it?). This method is used on the document object and takes the tag of the new element as a parameter. It returns the element created as an object (here stored in a variable called `pythonElement`).

```
const pythonElement = document.createElement("li"); // Create an li element
```

Setting element properties

Once the element's created and stored in a variable, you can add some detail to it (ID, class, text content, etc) by using the aforementioned DOM properties.

In the example, the element ID becomes `"python"` and its text content becomes `"Python"`.

```
// ...
pythonElement.id = "python";          // Define element ID
pythonElement.textContent = "Python"; // Define its text content
```

Inserting the element into the DOM

There are several techniques to insert a new node in the DOM. The most common is to call the `appendChild()` method on the element that will be the future parent of the new node. The new node is added to the end of the list of child nodes of that parent.

In our example, the new item is added as a new child of the `<ul>` tag identified by `"languages"`, after all the other children of this tag.

```
// ...
document.getElementById("languages").appendChild(pythonElement); // Insert th\
e new element into the DOM
```

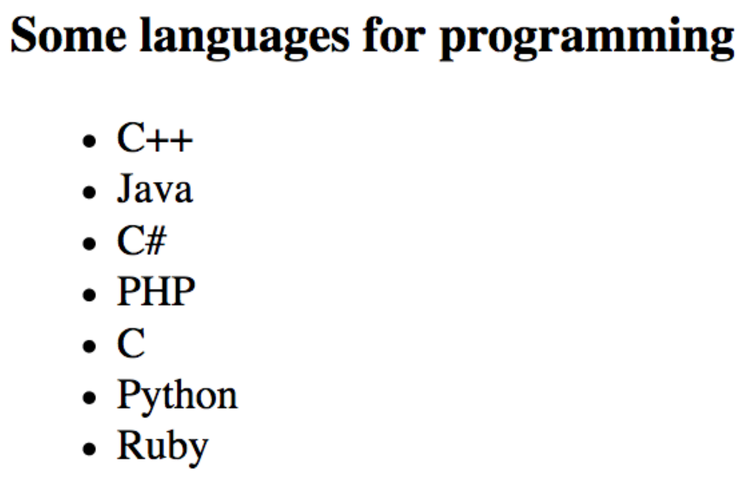
Variations on adding elements

Adding a textual node

Instead of using the `textContent` property to define the new element's textual content, you can create a textual node with the `createTextNode()` method. This node can then be added to the new element with `appendChild()`.

The following code demonstrates this possibility by inserting the Ruby language at the end of the list.

```
const rubyElement = document.createElement("li"); // Create an "li" element
rubyElement.id = "ruby"; // Define element ID
rubyElement.appendChild(document.createTextNode("Ruby")); // Define its text \
content
document.getElementById("languages").appendChild(rubyElement); // Insert the \
new element into the DOM
```



Some languages for programming

- C++
- Java
- C#
- PHP
- C
- Python
- Ruby

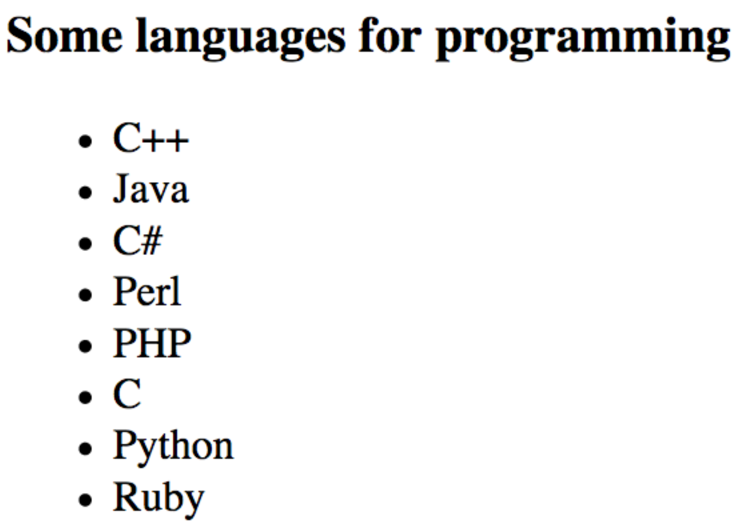
Execution result

Adding a node before another one

Sometimes, inserting a new node at the end of its parent's children list is not ideal. In that case, you can use the `insertBeforе()` method. Called on the future parent, this method takes as parameters the new node and the node before which the new one will be inserted.

As an example, here's how the Perl language could be inserted before PHP in the list.

```
const perlElement = document.createElement("li"); // Create an "li" element
perlElement.id = "perl"; // Define element ID
perlElement.textContent = "Perl"; // Define its text content
// Insert the new element before the "PHP" node
document.getElementById("languages").insertBefore(perlElement, document.getEl\
ementById("php"));
```


Some languages for programming

- C++
- Java
- C#
- Perl
- PHP
- C
- Python
- Ruby

Execution result

Determining the exact position of the new node

There is a method to more precisely define the position of inserted elements: `insertAdjacentHTML()`. Call it on an existing element and pass it the position and a string of HTML characters that represent the new content to be added. The new content's position should be either:

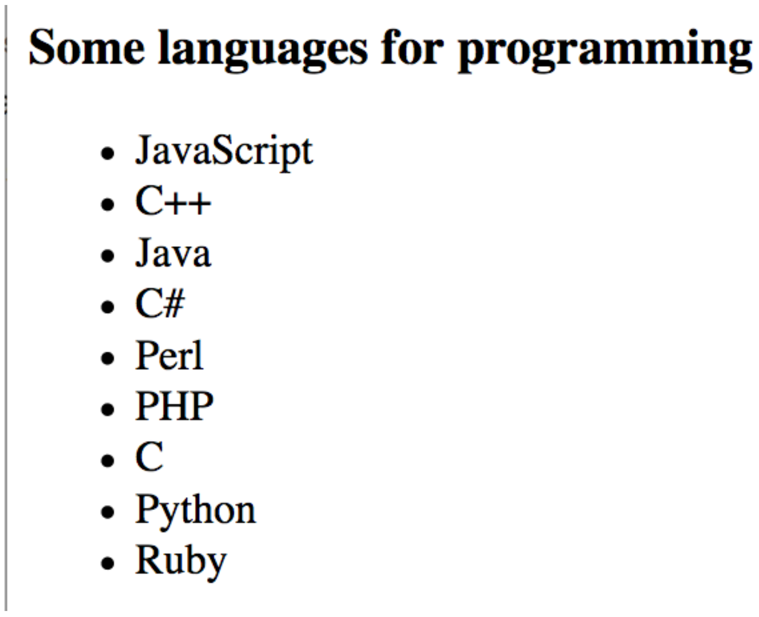
- `beforebegin`: before the existing element.
- `afterbegin`: inside the existing element, before its first child.
- `beforeend`: inside the existing element, after its last child.
- `afterend`: after the existing element.

Here's how these positions translate relative to an existing `<p>` tag.

```
<!-- beforebegin -->
<p>
  <!-- afterbegin -->
  foo
  <!-- beforeend -->
</p>
<!-- afterend -->
```

The following example uses `insertAdjacentHTML()` to add JavaScript at the top of the language list.

```
// Add an element to the beginning of a list
document.getElementById('languages').insertAdjacentHTML("afterBegin", '<li id\
="javascript">JavaScript</li>');
```



Some languages for programming

- JavaScript
- C++
- Java
- C#
- Perl
- PHP
- C
- Python
- Ruby

Execution result

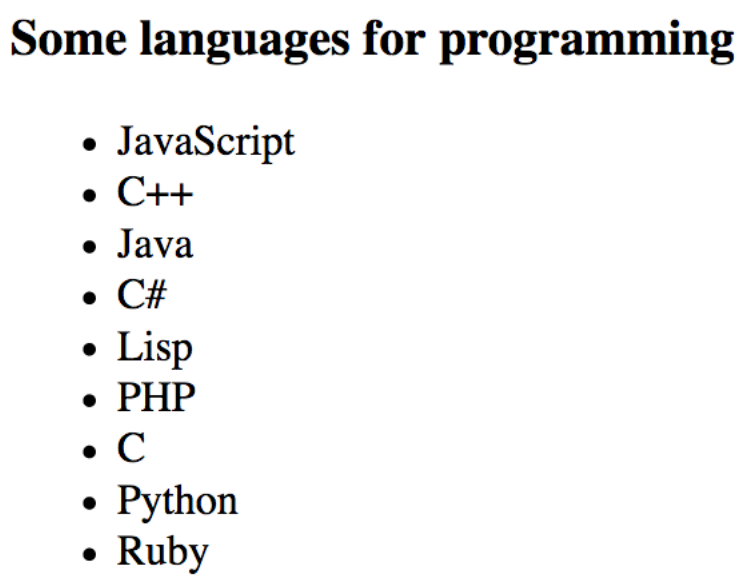
Replacing or removing nodes

Replacing a node

A DOM element can be replaced with the `replaceChild()` method. This replaces a child node of the current element with another node. The new node and node-to-be-replaced are passed as parameters (in that order).

The example shows replacing the Perl language with Lisp instead.

```
const lispElement = document.createElement("li"); // Create an li element
lispElement.id = "lisp";           // Define its ID
lispElement.textContent = "Lisp"; // Define its text content
// Replace the element identified by "perl" with the new element
document.getElementById("languages").replaceChild(lispElement, document.getEl\
ementById("perl"));
```


Some languages for programming

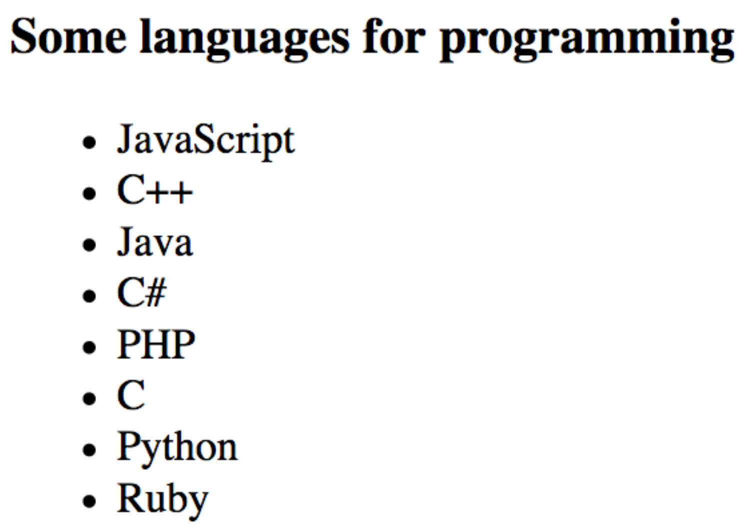
- JavaScript
- C++
- Java
- C#
- Lisp
- PHP
- C
- Python
- Ruby

Execution result

Removing a node

Lastly, you can delete a node thanks to a method called `removeChild()`, to which you'll pass the node-to-be-removed as a parameter.

```
// Remove the element with the "lisp" id
document.getElementById("languages").removeChild(document.getElementById("lis\
p"));
```


Some languages for programming

- JavaScript
- C++
- Java
- C#
- PHP
- C
- Python
- Ruby

Execution result

Styling elements

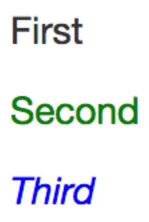
JavaScript not only lets you interact with your web page structure, but it also lets you change the style of elements. It's time to learn how.

Here is the example HTML content used in the next paragraphs.

```
<p>First</p>
<p style="color: green;">Second</p>
<p id="para">Third</p>
```

And here is the associated CSS **stylesheet**. The rules in a stylesheet determine the appearance of elements on a page. Here, the one element we're adjusting via CSS here is the element with the `para` ID. Its text will be blue and in italics.

```
#para {
    font-style: italic;
    color: blue;
}
```



Display result

The `style` property

DOM elements are equipped with a property called `style`, which returns an object representing the HTML element's `style` attribute. This object's properties match up to its CSS properties. By defining these properties with JavaScript, you're actually modifying the element's style.

The code below selects the page's first paragraph and modifies its text color and margins.

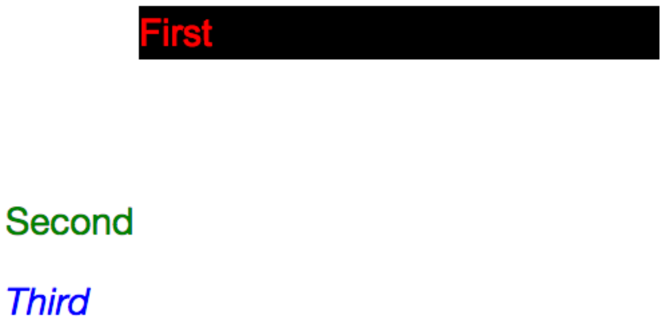
```
const paragraphElement = document.querySelector("p");
paragraphElement.style.color = "red";
paragraphElement.style.margin = "50px";
```

Compound CSS properties

Some CSS properties have compound names, meaning they're composed of two words (like `background-color`). To interact with these properties via JavaScript, you have to ditch the hyphen and capitalize the first letter of following words.

This example modifies the same paragraph element's `font-family` and `background-color` properties.

```
// ...
paragraphElement.style.fontFamily = "Arial";
paragraphElement.style.backgroundColor = "black";
```



First

Second

Third

Execution result

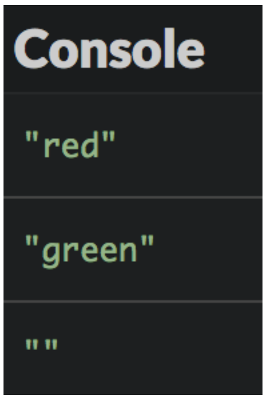
This naming convention, already encountered in previous chapters, is called camelCase[1].

You can see CSS properties and their JavaScript properties on the Mozilla Developer Network[2].

The limits of the `style` property

Let's try to display the text color of each of our example paragraphs.

```
const paragraphElements = document.getElementsByTagName("p");
console.log(paragraphElements[0].style.color); // "red"
console.log(paragraphElements[1].style.color); // "green"
console.log(paragraphElements[2].style.color); // Show an empty string
```



```
Console
"red"
"green"
""
```

Execution result

Why is the color of the third paragraph (blue) not showing?

[1] https://en.wikipedia.org/wiki/Camel_case
[2] https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Properties_Reference

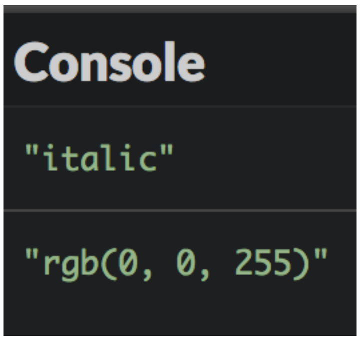
Because the `style` property used in this code only represents the `style` attribute of the element. Using this property, you cannot access style declarations defined elsewhere, for example in a CSS stylesheet. This explains why the third paragraph's style, defined externally, is not shown here.

Access element styles

A better solution for accessing element styles is to use a function called `getComputedStyle()`. Its takes a DOM node as a parameter and returns an object that represents its style. You can then see the different CSS properties of the object.

The following example will show the style properties of the third paragraph:

```
const paragraphStyle = getComputedStyle(document.getElementById("para"));
console.log(paragraphStyle.fontStyle); // "italic"
console.log(paragraphStyle.color);     // color blue in RGB values
```



```
Console
"italic"
"rgb(0, 0, 255)"
```

Execution result

> The blue color is represented as 3 color values: red, green, and blue (RGB). For each of these primary colors, values will always be between or equal to 0 and 255.

DOM manipulations and performance

Updating the DOM through JavaScript code causes the browser to compute the new page display. Frequent manipulations can lead to slowdowns and sub-par performance. As such, you should limit DOM access and update operations to a minimum.

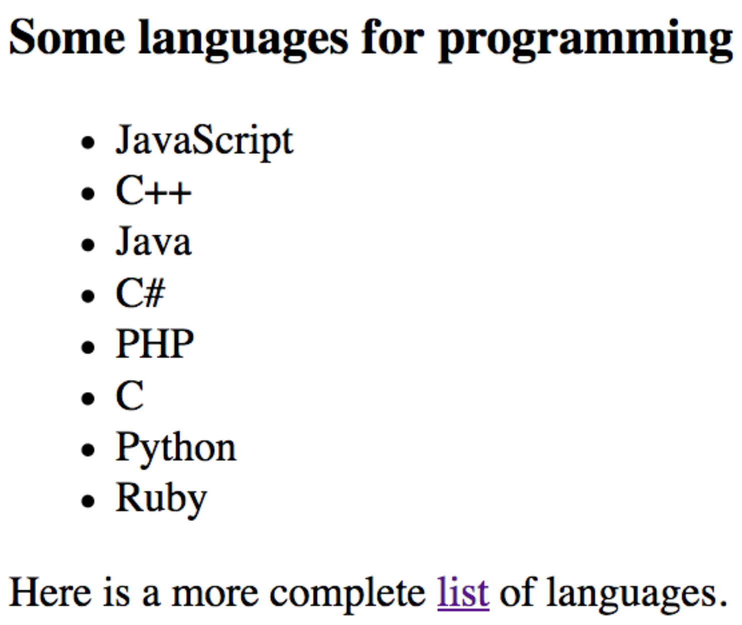
Creating and setting element properties *before* they're inserted into the DOM is a good way to preserve performance.

```
// Bad: DOM is updated multiple times
const newNode = document.createElement(...); // Create new element
parentNode.appendChild(newNode); // Add it to the DOM
newNode.id = ...; // Set some element properties
newNode.textContent = "...";
// ...

// Better: DOM is updated only once
const newNode = document.createElement(...); // Create new element
newNode.id = ...; // Set some element properties
newNode.textContent = "...";
// ...
parentNode.appendChild(newNode); // Add it to the DOM
```

Coding time!

Adding a paragraph

Improve the languages example to add a paragraph (`<p>` tag) containing a link (`<a>` tag) to the URL https://en.wikipedia.org/wiki/List_of_programming_languages.



Some languages for programming

- JavaScript
- C++
- Java
- C#
- PHP
- C
- Python
- Ruby

Here is a more complete list of languages.

Execution result

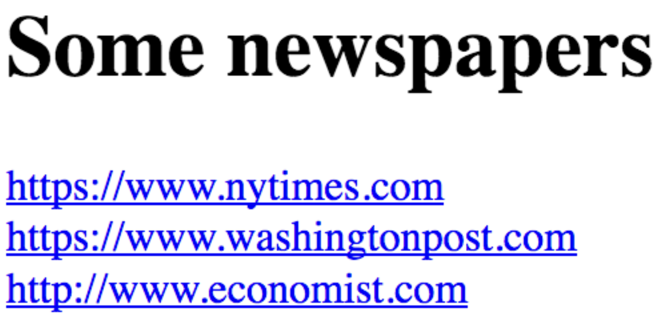
Newspaper list

Here is the HTML code of a web page.

```
<h3>Some newspapers</h3>
<div id="content"></div>
```

Write a program that show on this page a list of newspapers defined in a JavaScript array. Each link must be clickable.

```
// Newspaper list
const newspapers = ["https://www.nytimes.com", "https://www.washingtonpost.co\
m", "http://www.economist.com"];
```


Some newspapers

https://www.nytimes.com
https://www.washingtonpost.com
http://www.economist.com

Execution result

Mini-dictionary

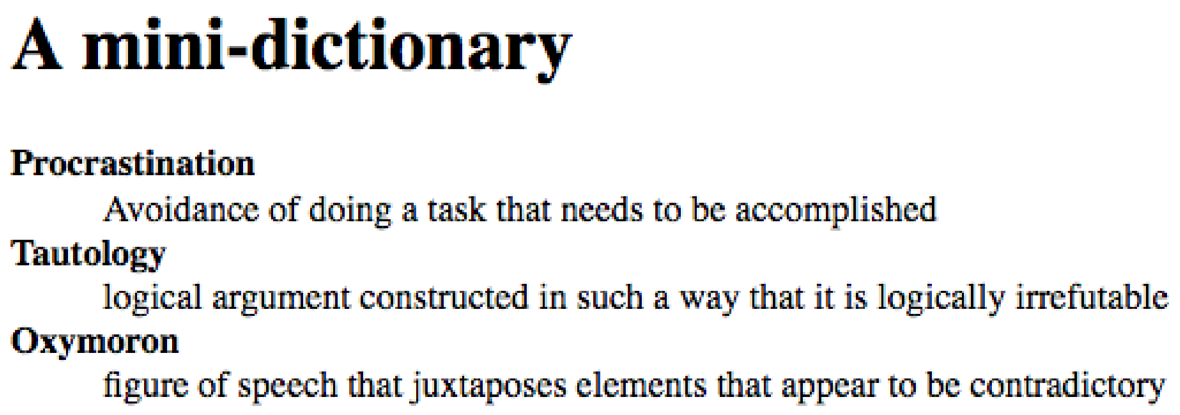
Here is the HTML code of a web page.

```
<h3>A mini-dictionary</h3>
<div id="content"></div>
```

Write a program that show on this page a list of terms and definitions defined in a JavaScript array.

```
const words = [{
  term: "Procrastination",
  definition: "Avoidance of doing a task that needs to be accomplished"
}, {
  term: "Tautology",
  definition: "logical argument constructed in such a way that it is logicall\
y irrefutable"
}, {
  term: "Oxymoron",
  definition: "figure of speech that juxtaposes elements that appear to be co\
ntradictory"
}];
```

Use the HTML `<dl>` tag to create the list (more on this tag[3]). Each term of the dictionary should be given more importance with a `<strong>` tag.



A mini-dictionary

Procrastination
Avoidance of doing a task that needs to be accomplished
Tautology
logical argument constructed in such a way that it is logically irrefutable
Oxymoron
figure of speech that juxtaposes elements that appear to be contradictory

Execution result

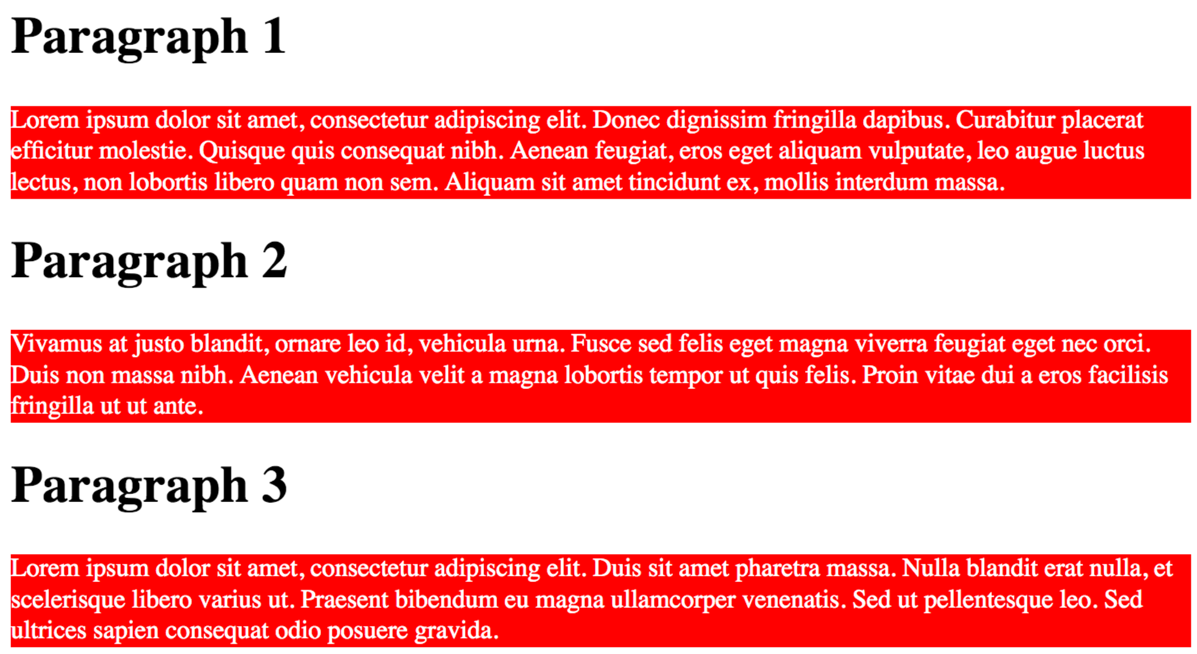
Updating colors

The following HTML content defines three paragraphs.

```
<h1>Paragraph 1</h1>
<div>Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec dignissim\
 fringilla dapibus. Curabitur placerat efficitur molestie. Quisque quis conse\
quat nibh. Aenean feugiat, eros eget aliquam vulputate, leo augue luctus lect\
us, non lobortis libero quam non sem. Aliquam sit amet tincidunt ex, mollis i\
nterdum massa.</div>

<h1>Paragraph 2</h1>
<div>Vivamus at justo blandit, ornare leo id, vehicula urna. Fusce sed felis \
eget magna viverra feugiat eget nec orci. Duis non massa nibh. Aenean vehicul\
a velit a magna lobortis tempor ut quis felis. Proin vitae dui a eros facilis\
is fringilla ut ut ante.</div>

<h1>Paragraph 3</h1>
<div>Lorem ipsum dolor sit amet, consectetur adipiscing elit. Duis sit amet p\
haretra massa. Nulla blandit erat nulla, et scelerisque libero varius ut. Pra\
esent bibendum eu magna ullamcorper venenatis. Sed ut pellentesque leo. Sed u\
ltrices sapien consequat odio posuere gravida.</div>
```

Write a program that asks the user for the new text color, then for the new background color. The page is then updated accordingly.

[3]https://developer.mozilla.org/en-US/docs/Web/HTML/Element/dl



Paragraph 1

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec dignissim fringilla dapibus. Curabitur placerat efficitur molestie. Quisque quis consequat nibh. Aenean feugiat, eros eget aliquam vulputate, leo augue luctus lectus, non lobortis libero quam non sem. Aliquam sit amet tincidunt ex, mollis interdum massa.

Paragraph 2

Vivamus at justo blandit, ornare leo id, vehicula urna. Fusce sed felis eget magna viverra feugiat eget nec orci. Duis non massa nibh. Aenean vehicula velit a magna lobortis tempor ut quis felis. Proin vitae dui a eros facilisis fringilla ut ut ante.

Paragraph 3

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Duis sit amet pharetra massa. Nulla blandit erat nulla, et scelerisque libero varius ut. Praesent bibendum eu magna ullamcorper venenatis. Sed ut pellentesque leo. Sed ultrices sapien consequat odio posuere gravida.

Execution result with red text on white background

Information about an element

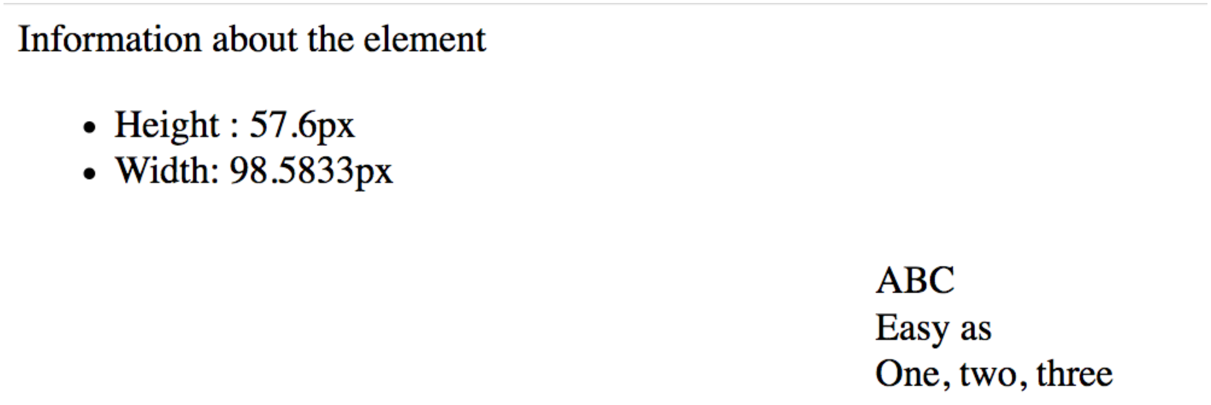
Here is this exercise’s HTML code.

```html
<div id="content">ABC
    <br>Easy as
    <br>One, two, three
</div>
<div id="infos"></div>
```

And the associated CSS stylesheet.

```css
#content {
    float: right;
    margin-top: 100px;
    margin-right: 50px;
}
```

Write a program that adds to the page a list showing the height and witdh of the element identified by “content”.


Information about the element

- Height : 57.6px
- Width: 98.5833px

ABC
Easy as
One, two, three

Execution result

16. React to events

To make a web page interactive, you have to respond to user actions. Let's discover how to do so.

TL;DR

- You can make a web page interactive by writing JavaScript code tied to **events** within the browser.
- Numerous types of events can be handled. Each event type is associated with an `Event` object that contains properties giving information about the event.
- `keypress`, `keydown` and `keyup` events let you react to keyboard-related events.
- `click`, `mousedown` and `mouseup` events let you react to mouse-related events.
- Page loading and closing are associated with the events `load` and `beforeunload` respectively.
- An event propagates within the DOM tree from its node of origin until the document root. This propagation can be interrupted with the `stopPropagation()` method.
- Calling the `preventDefault()` method on an `Event` object cancels the default behavior associated to the action that triggered the event.

Introduction to events

Up until now, your JavaScript code was executed right from the start. The execution order of statements was determined in advance and the only user interactions were data input through `prompt()` calls.

To add more interactivity, the page should react to the user's actions: clicking on a button, filling a form, etc. In that case, the execution order of statements is not determined in advance anymore, but depends on the user behavior. His actions trigger **events** that can be handled by writing JavaScript code.

This way of writing programs is called **event-driven programming**. It is often used by user interfaces, and more generaly anytime a program needs to interact with an user.

A first example

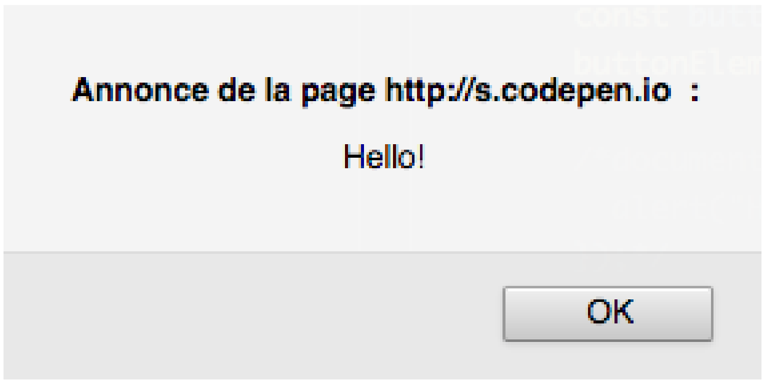
Here's some starter HTML code.

```html
<button id="myButton">Click me!</button>
```

And here's the associated JavaScript code.

```
function showMessage() {
  alert("Hello!");
}
// Access the button
const buttonElement = document.getElementById("myButton");
// Listen to the "click" event
buttonElement.addEventListener("click", showMessage);
```

Clicking on the web page button shows an `"Hello!` message.


Annonce de la page http://s.codepen.io :

Hello!

Execution result

Adding an event listener

Called on a DOM element, the `addEventListener()` method adds a **handler** for a particular event. This method takes as parameter the **event type** and the associated **function**. This function gets called whenever an event of the corresponding type appears for the DOM element.

The above JavaScript code could be rewritten more concisely using an anonymous function, for an identical result.

```
// Show a message when the user clicks on the button
document.getElementById("myButton").addEventListener("click", () => {
  alert("Hello!");
});
```

Removing an event listener

In some particular cases, you might want to stop reacting to an event on a DOM element. To achieve this, call the `removeEventListener()` on the element, massing as a parameter the function which used to handle the event.

This can only work if the handler function is not anonymous.

```
// Remove the handler for the click event
buttonElement.removeEventListener("click", showMessage);
```

The event family

Manay types of events can be triggered by DOM elements. Here are the main event categories.

Category	Examples
Keyboard events	Pressing or releasing a key
Mouse events	Clicking on a mouse button, pressing or releasing a mouse button, hovering over a zone
Window events	Loading or closing a page, resizing, scrolling
Form events	Changing focus on a form field, submitting a form

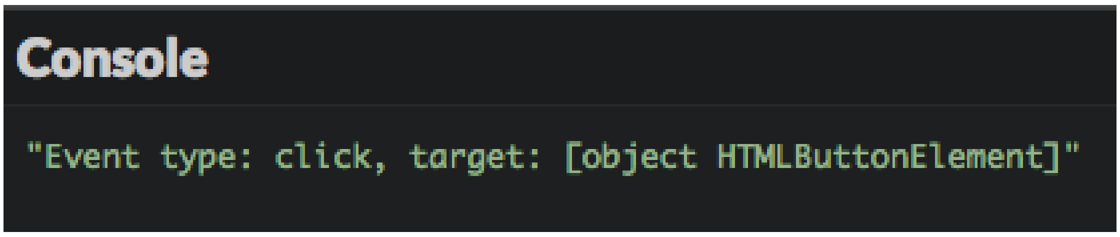
Every event is associated to an `Event` object which has both **properties** (informations about the event) and **methods** (ways to act on the event). This object can be used by the handler function.

Many properties of the `Event` object associated to an event depend on the event type. Some properties are always present, like `type` that returns the event type and `target` that return the event target (the DOM element that is the event source).

The `Event` object is passed as a parameter to the handler function. The following code uses it to show the event type and target in the console.

```
// Show event type and target when the user clicks on the button
document.getElementById("myButton").addEventListener("click", e => {
  console.log(`Event type: ${e.type}, target: ${e.target}`);
});
```

The parameter name chosen for the `Event` object is generaly `e` or `event`.


Console

```
"Event type: click, target: [object HTMLButtonElement]"
```

Execution result

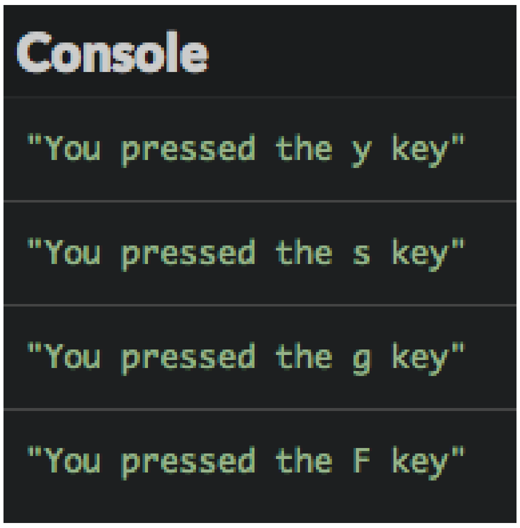
Reacting to common events

Key presses

The most common solution for reacting to key presses on a keyboard involves handling `keypress` events that happen on a web page (the DOM `body` element, which corresponds to the global variable called `document` in JavaScript).

The following example shows in the console the character assoaciated to a pressed key. Yhe character info is given by the `charCode` property of the `Event` object associated to the event. This property returns a numerical value (called **Unicode value**) that can be translated to a string value by the `String.FromCharCode()` method.

```
// Show the pressed character
document.addEventListener("keypress", e => {
    console.log(`You pressed the ${String.fromCharCode(e.charCode)} key`);
});
```

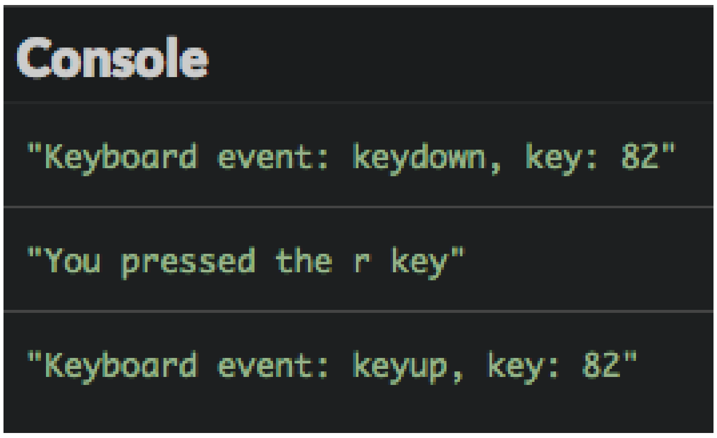


```
Console
"You pressed the y key"
"You pressed the s key"
"You pressed the g key"
"You pressed the F key"
```

Execution result

To manage the press and release of any key (not only the ones producing characters), you'll use the `keydown` and `keyup` events. This example uses the same function to manage two events. This time, the key's code is accessible in the `keyCode` property of the `Event` object.

```
// Show information on a keyboard event
function keyboardInfo(e) {
    console.log(`Keyboard event: ${e.type}, key: ${e.keyCode}`);
}

// Integrate this function into key press and release:
document.addEventListener("keydown", keyboardInfo);
document.addEventListener("keyup", keyboardInfo);
```



```
"Keyboard event: keydown, key: 82"
"You pressed the r key"
"Keyboard event: keyup, key: 82"
```

Execution result

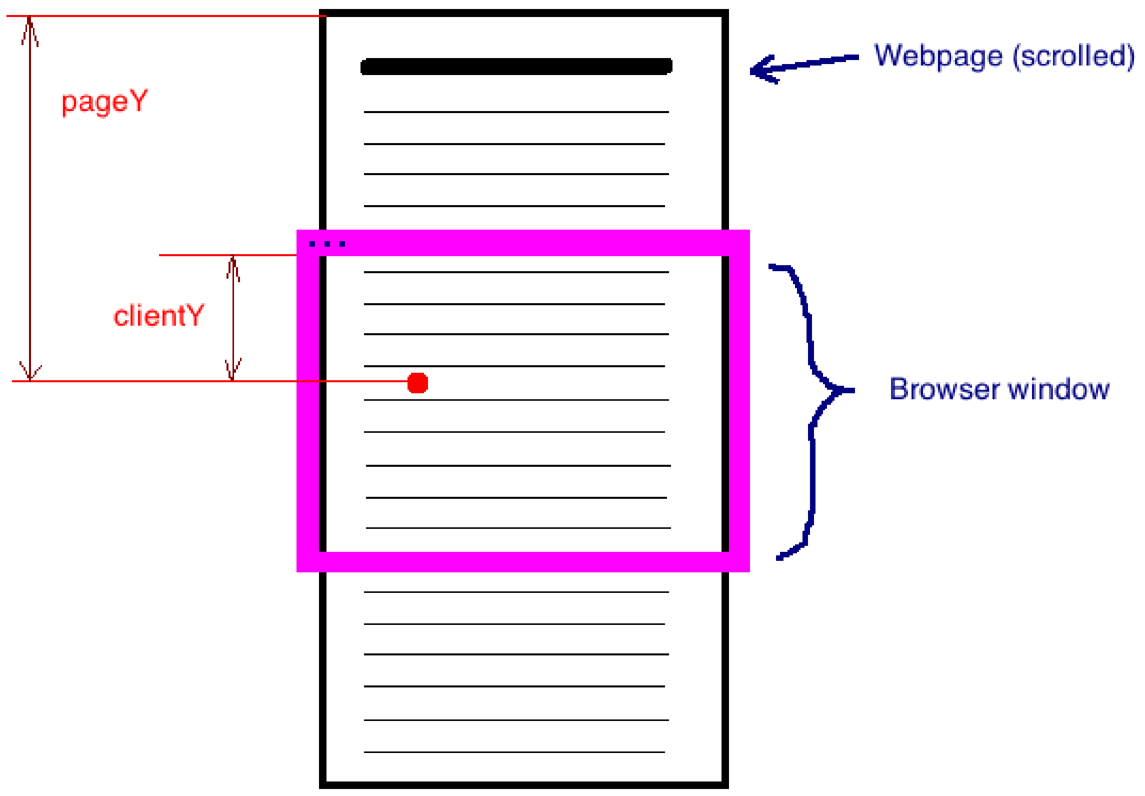
This results demonstrates that the launch order of keyboard-related events is as follows: `keydown -> keypress -> keyup`.

> The `keydown` is fired several times when a key is kept pressed.

Mouse clicks

Mouse clicks on any DOM element produce a event of the `click` type. Tactile interfaces like smartphones and tablets also have `click` events associated with buttons, which are kicked off by actually pressing a finger on the button.

The `Event` object associated with a `click` event has a `button` property which lets you know the mouse button used, as well as `clientX` and `clientY` properties that return the horizontal and vertical coordinates of the place where the click happened. These coordinates are defined relative to the page zone currently shown by the browser.

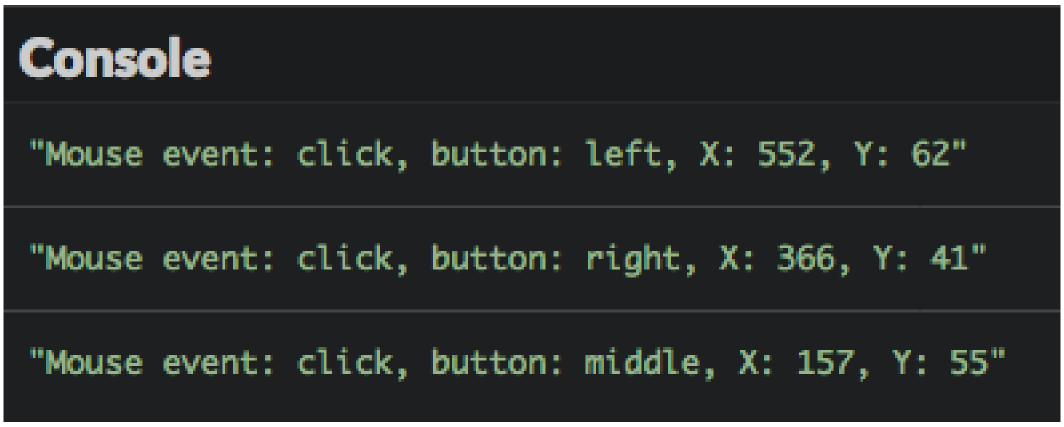


Difference between absolute and relative coordinates

The below code shows information on all click events that happen on a web page. The `mouseInfo()` function associated to the event uses another function, called `getMouseButton()`, to retrieve the clicked mouse button.

```
// Return the name of the mouse button
function getMouseButton(code) {
    let button = "unknown";
    switch (code) {
    case 0: // 0 is the code for the left mouse button
        button = "left";
        break;
    case 1: // 1 is the code for the middle mouse button
        button = "middle";
        break;
    case 2: // 2 is the code for the right button
        button = "right";
        break;
    }
    return button;
}
```

```
// Show info about mouse event
function mouseInfo(e) {
    console.log(`Mouse event: ${e.type}, button: ${getMouseButton(e.button)},\
 X: ${e.clientX}, Y: ${e.clientY}`);
}

// Add mouse click event listener
document.addEventListener("click", mouseInfo);
```

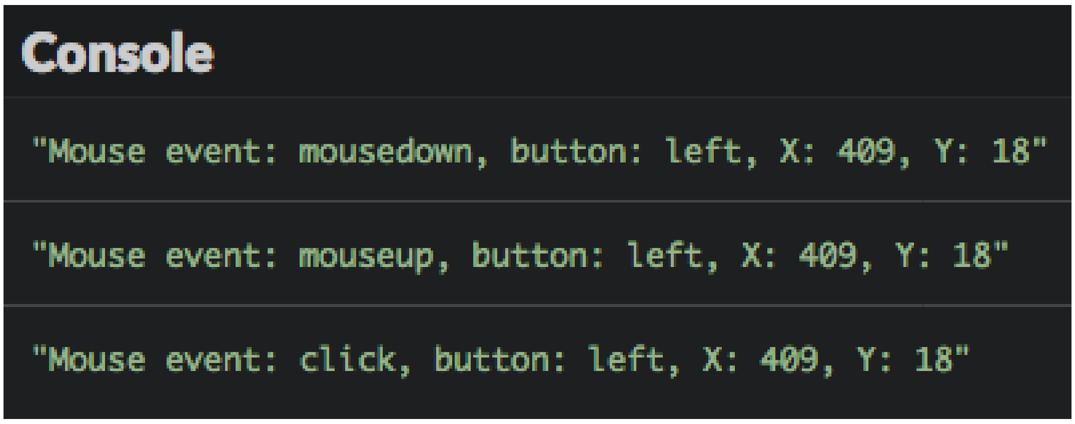


```
Console
"Mouse event: click, button: left, X: 552, Y: 62"
"Mouse event: click, button: right, X: 366, Y: 41"
"Mouse event: click, button: middle, X: 157, Y: 55"
```

Execution result

You can use `mousedown` and `mouseup` events similarly to `keydown` and `keyup` to deal with mouse buttons' press and release events. The code below associates the same handler to these two events.

```
// Handle mouse button press and release
document.addEventListener("mousedown", mouseInfo);
document.addEventListener("mouseup", mouseInfo);
```



```
Console
"Mouse event: mousedown, button: left, X: 409, Y: 18"
"Mouse event: mouseup, button: left, X: 409, Y: 18"
"Mouse event: click, button: left, X: 409, Y: 18"
```

Execution result

The appearance order for mouse-related events is: `mousedown` -> `mouseup` -> `click`.

Page loading

Depending on how complex it is, a web page can take time to be entirely loaded by the browser. You can add an event listener on the `load` event produced by the `window` object (which represent the brower window) to know when this happens. This avoids messy situations where JavaScript interacts with pages that aren't fully loaded.

The following code displays a message in the console once the page is fully loaded.

```
// Web page loading event
window.addEventListener("load", e => {
    console.log("The page has been loaded!");
});
```

Page closing

You sometimes want to react to page closing. Closing happens when the user closes the tab displaying the page or navigates to another page in this tab. A frequent use case consists of showing a confirmation dialog to the user. Handling page closing is done by adding a handler for the `beforeunload` event on the `window` object.

```
// Handle page closing
window.addEventListener("beforeunload", e => {
  const message = "Should you stay or should you go?";
  // Standard way of showing a confirmation dialog
  e.returnValue = message;
  // Browser-specific way of showing a confirmation dialog
  return message;
});
```

Setting the value of the `returnValue` property on the `Event` object is the standard way of triggering a confirmation dialog showing this value. However, some browsers use the return value of the event listener instead. The previous code associate the two techniques to be universal.

Go farther with events

Event propagation

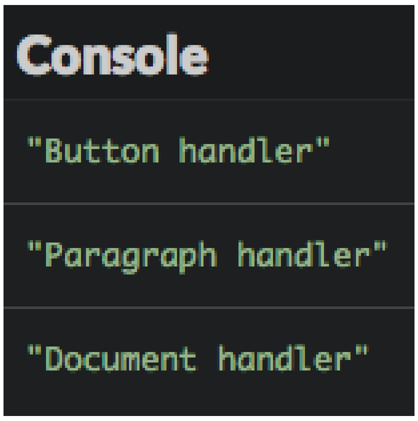
The DOM represents a web page as a hierarchy of nodes. Events triggered on a child node are going to get triggered on the parent node, then the parent node of the parent node, up until the root of the DOM (the `document` variable). This is called **event propagation**.

To see propagation in action, use this HTML code to create a small DOM hierachy.

```html
<p id="para">A paragraph with a <button id="propa">button</button> inside</p>
```

Here’s the complementary JavaScript code. It adds `click` event handlers on the button, its parent (the paragraph), and the parent of that too (the root of the DOM).

```js
// Click handler on the document
document.addEventListener("click", e => {
    console.log("Document handler");
});
// Click handler on the paragraph
document.getElementById("para").addEventListener("click", e => {
    console.log("Paragraph handler");
});
// Click handler on the button
document.getElementById("propa").addEventListener("click", e => {
    console.log("Button handler");
    e.stopPropagation(); // Stop the event propagation
});
```



```
Console
"Button handler"
"Paragraph handler"
"Document handler"
```

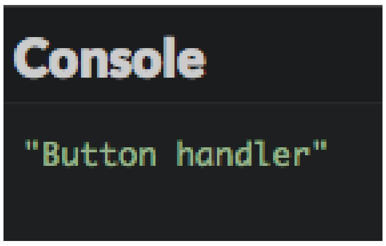
Execution result

The result in the browser console demonstrates the propagation of `click` events from the button up to the document level. You clicked the button, which means you also clicked the paragraph, which means you also clicked the document.

But maybe you only want an event to kick off once the button is clicked and not count its larger ecosystem? Event propagation can be interrupted at any moment by calling the `stopPropagation()` method on the `Event` object from an event handler. This is useful to avoid the same event being handled multiple times.

Adding a line in the button’s click handler prevents the `click` event from propagating everywhere in the DOM tree.

```
// Click handler on the button
document.getElementById("propa").addEventListener("click", e => {
    console.log("Button handler");
    e.stopPropagation(); // Stop the event propagation
});
```



```
Console

"Button handler"
```

Execution result

Cancelling the default behavior of an event

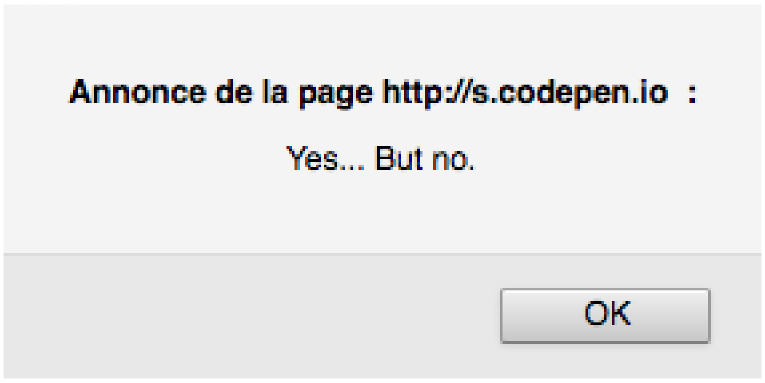
Most of the user actions on a page are associated to a default behavior. Clicking on a link navigates to the link target, clicking anywhere with the right mouse button show a contextual menu, etc. Cancelling a default behavior is possible by calling the `preventDefault()` method on the `Event` object in an event handler.

Let's use the following HTML code to see this possibility in action.

```
<p>Time on your hands? <a id="forbidden" href="https://9gag.com/">Click here<\
/a></p>
```

```
// Handling clicking on the forbidden link
document.getElementById("forbidden").addEventListener("click", e => {
  alert("Yes... But no.");
  e.preventDefault(); // Cancels the default behavior
});
```

Now clicking on the links shows a dialog instead of navigating to its target.



Annonce de la page http://s.codepen.io :

Yes... But no.

Execution result

Coding time!

Counting clicks

Start with the following HTML content.

```
<button id="myButton">Click me!</button>
<p>You clicked on the button <span id="clickCount">0</span> times</p>
<button id="desactivate">Désactivate counting</button>
```

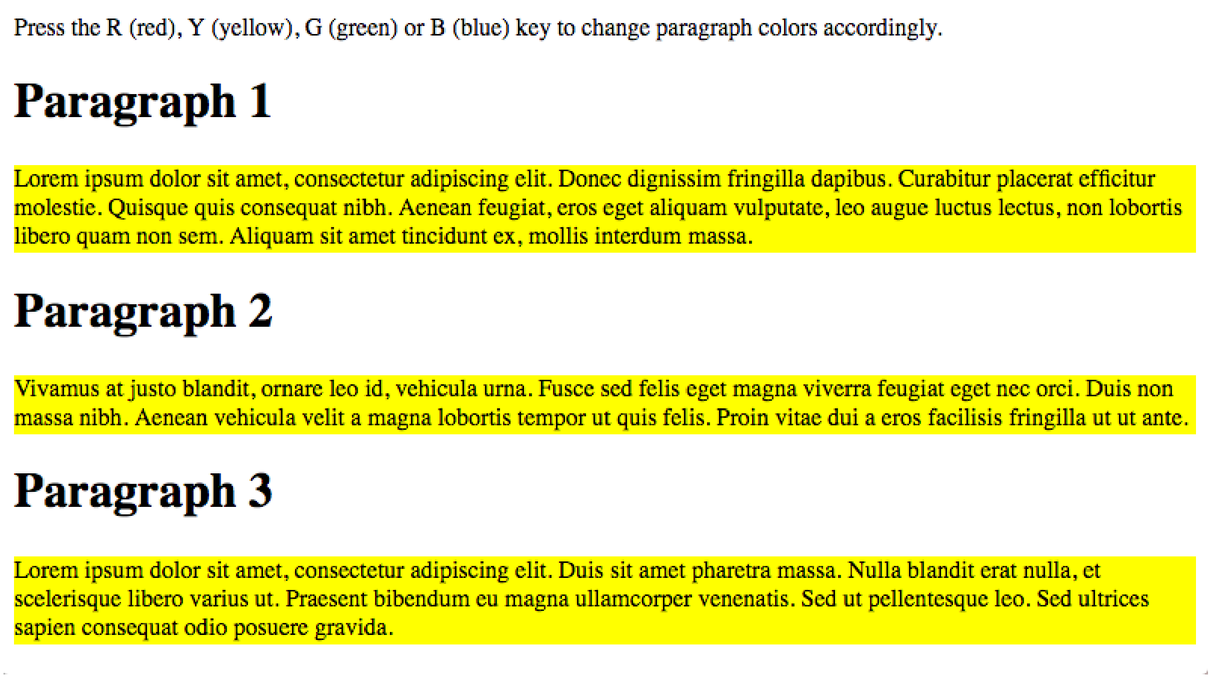
Write the JavaScript code that counts the number of clicks on the `myButton` button by updating the `clickCount` element. The `desactivate` button stop the counting.

Changing colors

Here is some HTML content to start with.

```
<p>Press the R (red), Y (yellow), G (green) or B (blue) key to change paragra\
ph colors accordingly.</p>

<h1>Paragraph 1</h1>
<div>Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec dignissim\
 fringilla dapibus. Curabitur placerat efficitur molestie. Quisque quis conse\
quat nibh. Aenean feugiat, eros eget aliquam vulputate, leo augue luctus lect\
us, non lobortis libero quam non sem. Aliquam sit amet tincidunt ex, mollis i\
nterdum massa.</div>

<h1>Paragraph 2</h1>
<div>Vivamus at justo blandit, ornare leo id, vehicula urna. Fusce sed felis \
eget magna viverra feugiat eget nec orci. Duis non massa nibh. Aenean vehicul\
a velit a magna lobortis tempor ut quis felis. Proin vitae dui a eros facilis\
is fringilla ut ut ante.</div>

<h1>Paragraph 3</h1>
<div>Lorem ipsum dolor sit amet, consectetur adipiscing elit. Duis sit amet p\
haretra massa. Nulla blandit erat nulla, et scelerisque libero varius ut. Pra\
esent bibendum eu magna ullamcorper venenatis. Sed ut pellentesque leo. Sed u\
ltrices sapien consequat odio posuere gravida.</div>
```

Write the associated JavaScript code that update background color of all `div` tags according to the key (R, Y, G or B) pressed by the user.


Press the R (red), Y (yellow), G (green) or B (blue) key to change paragraph colors accordingly.

Paragraph 1

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec dignissim fringilla dapibus. Curabitur placerat efficitur molestie. Quisque quis consequat nibh. Aenean feugiat, eros eget aliquam vulputate, leo augue luctus lectus, non lobortis libero quam non sem. Aliquam sit amet tincidunt ex, mollis interdum massa.

Paragraph 2

Vivamus at justo blandit, ornare leo id, vehicula urna. Fusce sed felis eget magna viverra feugiat eget nec orci. Duis non massa nibh. Aenean vehicula velit a magna lobortis tempor ut quis felis. Proin vitae dui a eros facilisis fringilla ut ut ante.

Paragraph 3

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Duis sit amet pharetra massa. Nulla blandit erat nulla, et scelerisque libero varius ut. Praesent bibendum eu magna ullamcorper venenatis. Sed ut pellentesque leo. Sed ultrices sapien consequat odio posuere gravida.

Execution result

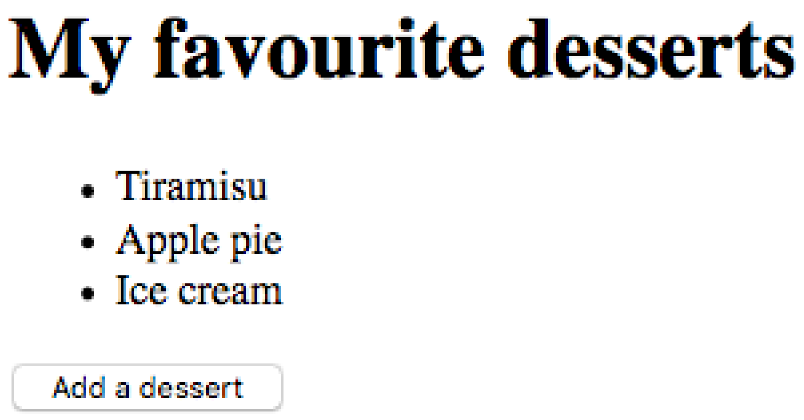
A dessert list

The following HTML code defines a list of desserts, empty for now.

```html
<h1>My favourite desserts</h1>

<ul id="desserts">
</ul>

<button id="addButton">Add a dessert</button>
```

Write the JavaScript code that adds a new dessert to the list when the user clicks on the button. The dessert name is chosen by the user.

Bonus points for adding the possibility of changing a dessert's name when clicking on it.



Execution result

Interactive quiz

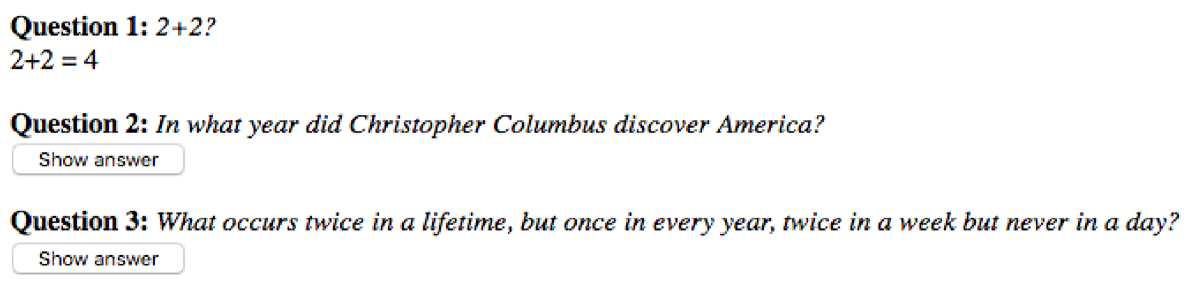
Here is the starter HTML code.

```html
<div id="content"></div>
```

And the associated JavaScript code that defines a question list.

```javascript
// List of questions (statement + answer)
const questions = [
{
    statement: "2+2?",
    answer: "2+2 = 4"
},
{
    statement: "In what year did Christopher Columbus discover America?",
    answer: "1492"
},
{
    statement: "What occurs twice in a lifetime, but once in every year, twic\
e in a week but never in a day?",
    answer: "The E letter"
}
];
```

Complete this code to display the questions in the `<div>` element of the page, with a `"Show the answer"` button next to each question. Clicking this button replaces it with the answer for this question.


Question 1: *2+2?*
2+2 = 4

Question 2: *In what year did Christopher Columbus discover America?*

Question 3: *What occurs twice in a lifetime, but once in every year, twice in a week but never in a day?*

Execution result

Conclusion

Acknowledgments

This book was built upon two online courses I wrote for the French EdTech startup OpenClassrooms[1]:

- Learn to code with JavaScript[2] (Apprenez à coder avec JavaScript[3])
- Use JavaScript in your web projects[4] (Créez des pages web interactives avec JavaScript[5])

Thanks to Jessica Mautref[6] for her watchful eye during the writing process, and to Emily Reese[7] for the initial English translation. Both also contributed many good ideas.

I've been inspired by other authors who decided to publish their books in an open way: Kyle Simpson[8], Nicholas C. Zakas[9], Axel Rauschmayer[10] and Marijn Haverbeke[11].

Thanks to everyone who improved the book content through contributions: Gilad Penn, NewMountain, Emre Akbudak, opheron. Special thanks to Theo Armour.

Cover logo: Creative blue swirl[12] by Free Logo Design[13].

[1]https://openclassrooms.com

[2]https://openclassrooms.com/courses/learn-the-basics-of-javascript

[3]https://openclassrooms.com/courses/apprenez-a-coder-avec-javascript

[4]https://openclassrooms.com/courses/use-javascript-on-the-web

[5]https://openclassrooms.com/courses/creez-des-pages-web-interactives-avec-javascript

[6]https://www.linkedin.com/in/jessicamautref

[7]https://www.linkedin.com/in/eclairereese

[8]https://github.com/getify

[9]https://www.nczonline.net/

[10]http://dr-axel.de/

[11]http://marijnhaverbeke.nl/

[12]http://www.logoopenstock.com/logo/preview/64186/creative-blue-swirl-logo-design

[13]http://www.free-logodesign.com/